

А.Я. Архангельский

100 компонентов общего назначения библиотеки Delphi 5

Подробное
рассмотрение
свойств и методов
компонентов

Сравнительный
анализ функций
и возможностей
компонентов

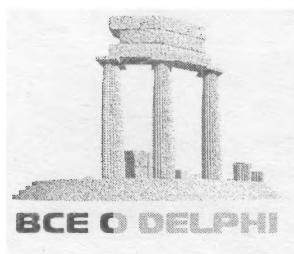
Рекомендации
по применению

Современный
дизайн программ
и организация
управления



БИНОМ

ВСЕ О DELPHI



А.Я. Архангельский

100 компонентов общего назначения библиотеки Delphi 5



Москва

ЗАО «Издательство БИНОМ»

1999

УДК 004.43
ББК 32.973-018.1
А87

Архангельский А. Я.

100 компонентов общего назначения библиотеки Delphi 5 — М.: ЗАО «Издательство БИНОМ», 1999. — 272 с.: ил. (Все о Delphi).

В книге рассмотрено 100 компонентов и классов Delphi 5 — т.е. практически все компоненты общего назначения, кроме специализированных компонентов, связанных с данными, и компонентов для работы с Интернет, которые требуют отдельного рассмотрения. Проведен сравнительный анализ функций и возможностей компонентов, даны рекомендации по их применению. Приводятся многочисленные примеры.

Книга рассчитана на пользователей любой квалификации: от начинающих до опытных разработчиков.

Содержание

От редактора серии «Все о Delphi»	9
От автора	12
Глава 1. Основы разработки прикладных программ в Delphi	15
1.1 Интегрированная среда разработки Delphi 5	15
1.2 Структура кода модуля	22
1.3. Программный доступ к свойствам и методам объектов.	27
Глава 2. Библиотека компонентов Delphi	31
2.1 Организация библиотеки компонентов	31
2.2. Страницы библиотеки компонентов	34
2.2.1 Страница Standard	34
2.2.2 Страница Additional	36
2.2.3 Страница Win32	38
2.2.4 Страница System	40
2.2.5 Страница QReport	41
2.2.6 Страница Dialogs	43
2.2.7 Страница Win3.1.	45
2.2.8 Страница Samples	46
2.2.9 Страница ActiveX	47
2.3. Настройка палитры компонентов	48
2.4. Создание и запись в библиотеку шаблонов компонентов и групп компонентов.	49
2.5 Некоторые итоги	52
Глава 3. Компоненты ввода и отображения текстовой, цифровой и иерархической информации	53
3.1 Font — шрифт отображения текста в компонентах	53
3.2 Вывод текста на канву Canvas	57
3.3 Компоненты ввода и отображения текстовой информации	62
3.3.1 Перечень компонентов ввода и отображения текстовой информации	62

3.3.2	Отображение текста в надписях компонентов Label, StaticText и Panel	64
3.3.3	Окна редактирования Edit и MaskEdit	67
3.3.4	Многострочные окна редактирования Memo и RichEdit	73
3.3.5	Компоненты выбора из списков — ListBox, CheckListBox и ComboBox	79
3.3.6	Таблица строк — компонент StringGrid	82
3.4	Ввод и отображение чисел, дат и времени	84
3.4.1	Перечень компонентов ввода и отображения чисел, дат и времени	84
3.4.2	Ввод и отображение целых чисел — компоненты UpDown и SpinEdit	85
3.4.3	Ввод и отображение дат и времени — компоненты DateTimePicker, MonthCalendar, Calendar	87
3.4.4	Страницы Excel — компонент F1Book	89
3.5	Обеспечение синтаксически правильного ввода текстовых и цифровых данных	91
3.6	Компоненты отображения заголовков и иерархических данных	92
3.6.1	Перечень компонентов	92
3.6.2	Секционированное отображение текстов — компоненты HeaderControl и Header	94
3.6.3	Компоненты отображения иерархических данных — TreeView, Outline, ListView	95
Глава 4.	Графика, печать, мультимедиа	107
4.1	Перечень компонентов отображения графической информации	107
4.2	Отображение графики на канве Canvas	107
4.3	Компоненты Image и PaintBox	116
4.4	Таблицы изображений — компоненты DrawGrid и StringGrid	123
4.5	Компонент Shape	124
4.6	Построение графиков и диаграмм	125
4.6.1	Компонент Chart	125
4.6.2	Компоненты страницы ActiveX — F1Book, Chartfx, Graph	132
4.7	Печать	136
4.7.1	Методы компонентов, обеспечивающие печать	136
4.7.2	Печать с помощью объекта Printer	138
4.7.3	Подготовка и печать отчетов с помощью QuickReport — компоненты QuickRep, QRSubDetail, QRLabel, QRExpr, QRSysData, QRMemo, QRRichText, QRShape, QRImage	139
4.8	Мультимедиа и анимация	151

4.8.1 Отображение хода выполнения длительных операций — компоненты <code>ProgressBar</code> и <code>Gauge</code>	151
4.8.2 Воспроизведение немых видео клипов — компонент <code>Animate</code>	153
4.8.3 Универсальный проигрыватель <code>MediaPlayer</code>	157
Глава 5. Кнопки, индикаторы, управляющие элементы	163
5.1 Общая характеристика	163
5.2 Управляющие кнопки <code>Button</code> и <code>BitBtn</code>	165
5.3 Кнопка с фиксацией <code>SpeedButton</code>	168
5.4 Группы радиокнопок — компоненты <code>RadioGroup</code> , <code>RadioButton</code> и <code>GroupBox</code>	170
5.5 Индикаторы <code>CheckBox</code> и <code>CheckListBox</code>	172
5.6 Ползунки и полосы прокрутки — компоненты <code>TrackBar</code> и <code>ScrollBar</code>	173
5.7 Таймер — компонент <code>Timer</code>	176
Глава 6. Компоненты — меню	179
6.1 Главное меню — компонент <code>MainMenu</code>	179
6.2 Контекстное всплывающее меню — компонент <code>PopupMenu</code>	185
6.3 Горячие клавиши — компонент <code>HotKey</code>	186
Глава 7. Панели и компоненты внешнего оформления	191
7.1 Общая характеристика	191
7.2 Панели общего назначения — компоненты <code>Panel</code> , <code>GroupBox</code> , <code>Bevel</code> , <code>ScrollBox</code> , <code>Splitter</code>	193
7.3 Многостраничные панели — компоненты <code>TabControl</code> , <code>PageControl</code> , <code>TabSet</code> , <code>TabbedNoteBook</code> , <code>NoteBook</code>	198
7.4 Инструментальные панели — компоненты <code>ToolBar</code> и <code>PageScroller</code>	202
7.5 Перестраиваемые панели — компоненты <code>CoolBar</code> и <code>ControlBar</code>	206
7.6 Полоса состояния <code>StatusBar</code>	209
7.7 Фреймы	212

Глава 8. Системные диалоги	223
8.1 Общая характеристика компонентов — диалогов. . .	223
8.2 Диалоги открытия и сохранения файлов — компоненты OpenFileDialog, SaveDialog, OpenPictureDialog, SavePictureDialog	225
8.3 Фрагменты диалогов — компоненты DriveComboBox, DirectoryListBox, FilterComboBox, FileListBox и DirectoryOutline. . . .	232
8.4 Диалог выбора шрифта — компонент FontDialog. . .	235
8.5 Диалог выбора цвета — компонент ColorDialog . . .	238
8.6 Диалоги печати и установки принтера — компоненты PrintDialog и PrinterSetupDialog	239
8.7 Диалоги поиска и замены текста — компоненты FindDialog и ReplaceDialog.	241
Глава 9. Компоненты организации управления приложением	247
9.1 Общая характеристика.	247
9.2 Диспетчеризация событий — компоненты, связанные с ActionList	248
9.3 Список изображений — компонент ImageList	252
9.4 Приложение — компонент ApplicationEvents и объект Application	255
9.5 Объект экрана — Screen	263

От редактора серии «Все о Delphi»

Перед вами вторая книга серии «Все о Delphi». Поскольку для вас она, возможно, первая из этой серии, на которую вы обратили внимание, я позволю себе повторить то, о чем писал в первой книге: рассказать, что это за серия, зачем ее придумали и чего от нее можно ожидать.

Delphi — чрезвычайно мощная система разработки прикладных программ для Windows. Она все увереннее завоевывает ведущие позиции в нашей стране как среди профессиональных программистов, так и среди людей, никогда профессионально программированием не занимавшихся. Профессионалы используют Delphi для построения сложных приложений с распределенными базами данных и для многих других целей. Специалисты самых разных специальностей (не программисты) создают с помощью Delphi небольшие прикладные программы для решения своих профессиональных задач и радуются, что, не владея всерьез программированием, могут создавать интерфейс, неотличимый от привычного интерфейса Windows. Студенты (не программисты) выполняют с помощью Delphi свои курсовые и дипломные проекты. Сейчас уже есть даже средние школы, в которых начинают знакомить с работой в среде Delphi.

Такой несомненно большой интерес к Delphi разных категорий пользователей породил, конечно, спрос на соответствующую литературу. И спрос этот удовлетворяется. Выпущено множество книг по Delphi, от самоучителей, до энциклопедий. Но, к сожалению, очень часто читатель не может найти в купленной, подчас весьма дорогой книге ответы на все интересующие его вопросы. И это не случайно. Delphi — очень большая, многофункциональная система, области ее применения разнообразны, соответственно разнообразны и вопросы, интересующие читателя. Все это изложить в одной книге просто невозможно.

Но давайте представим себе невозможное: кто-то написал книгу о Delphi, содержащую все, что может интересовать пользователей. Она содержит и методику работы с системой (причем изложенную как для начинающих, так и для специалистов, включая язык турбо ассемблер — ведь без него не разберешься в окне CPU, встроенном в оболочку Delphi), и полное описание библиотеки компонентов с многочисленными примерами, и описание языка Object Pascal, на котором построена Delphi, и полный

справочный материал по всем функциям Object Pascal и API Windows, и методику работы с разными базами данных, и методику работы с Интернет, и разработку собственных компонентов, и многое — многое другое. Такая книга по самым скромным подсчетам должна содержать несколько тысяч страниц. О цене подобной книги скромно умолчим. Подумаем о другом — понравится ли она читателю. Наверняка нет! Ведь у каждого имеется определенный круг профессиональных интересов. Кого-то совершенно не интересует разработка приложений для Интернет, кто-то никогда в жизни не будет разрабатывать собственных компонентов, так как в Delphi их и так имеется великое множество, кто-то не собирается углубляться в системное программирование, не собирается работать с распределенными базами данных и т.д. Так что практически любой читатель, купив такую гипотетическую всеобъемлющую книгу, выбросит на ветер значительную часть своих денег — на тот материал, который ему не нужен ни сейчас, ни когда-либо в будущем. К тому же учтите, что новые версии Delphi появляются ежегодно. Так что через год эта книга устареет. И что делать? Создавать еще одну подобную книгу — монстра?

Серия «Все о Delphi», которая начала выпускаться, призвана решить эту казалось бы неразрешимую задачу. Она будет состоять из значительного числа (более 10) сравнительно небольших (и, значит, не особо дорогих) книг, в каждой из которых полностью рассматривается тот или иной вопрос, связанный с Delphi. Каждую из этих книг можно читать независимо от других. А все вместе они с течением времени составят ту всеобъемлющую книгу о Delphi, о которой шла речь выше. Но преимущество серии в том, что каждый может покупать только те выпускаемые в ней книги, которые входят в круг его интересов. Таким образом, через некоторое время каждый интересующийся сможет составить из книг серии свою собственную библиотеку Delphi, куда войдет все, что ему нужно, и не войдет лишнего материала, не представляющего для него интереса. Легко решится и вопрос с новыми версиями Delphi. Можно будет выпускать книги, где излагается в основном только то новое, что появится в новых версиях, без неоправданных повторов уже известного читателю.

Одна из книг серии, посвященная интегрированной среде разработки Delphi, уже вышла в свет. В ближайшее время планируется издать еще ряд книг серии, которые, вероятно, будут полезны всем читателям — и профессионалам, и начинающим. Это книги, посвященные методике работы с Delphi, методологии создания приложений для Windows, работе с базами данных, языку Object Pascal. В дальнейшем пойдут книги, содержащие более

специальные вопросы. И тут хотелось бы обратиться к читателям за помощью. Давайте создавать нашу серию вместе. Среди читателей, наверное, есть немало потенциальных авторов будущих книг серии. Если вы хорошо освоили какие-то аспекты Delphi, если у вас есть интересные применения Delphi и вы готовы поделиться своими знаниями с другими, мы с удовольствием обсудим ваши предложения. Телефон редакции — (095) 973-9062.

От автора

Эта книга — вторая в серии, посвященной Delphi — популярной среди разработчиков системе визуального объектно-ориентированного проектирования прикладных программ для Windows. Delphi — превосходный инструмент, с помощью которого и начинающий пользователь, и программист-профессионал могут создавать одинаково профессионально выглядящий интерфейс пользователя к прикладным программам самых различных классов.

В данной книге описаны компоненты библиотеки Delphi 5. Описание скомпоновано не по страницам библиотеки, а по назначению компонентов и по выполняемым ими функциям. Это позволяет параллельно рассматривать сходные компоненты, сравнивать их возможности и давать рекомендации по их применению.

Описаны также некоторые классы и типы Delphi 5, которые формально не являются компонентами, поскольку не включены в палитру библиотеки, но без которых изложение было бы неполным. Например, невозможно описывать компоненты отображения графической информации, не рассмотрев канву Canvas, перо Pen, кисть Brush. А такие объекты, как принтер Printer, приложение Application или экран Screen, фактически являются невидимыми компонентами и не включены в страницы библиотеки только потому, что в каждой прикладной программе они имеются всего в одном экземпляре.

К сожалению, в небольшой книге невозможно описать все компоненты библиотеки. Поэтому пришлось пойти на самоограничение — отказаться от описания компонентов, предназначенных для работы с базами данных и с Интернет. Применение этих компонентов связано со спецификой соответствующих прикладных программ, которая требует отдельного и детального рассмотрения. Эти компоненты будут описаны в будущих книгах серии «Все о Delphi».

Данная книга предназначена для пользователей различной квалификации — от начинающих до опытных специалистов. Начинаящий найдет в ней простое изложение методики работы с различными компонентами. Специально для этой категории читателей введена небольшая первая глава и отдельные разделы

второй главы, поясняющие общие принципы работы с Delphi 5. А опытному пользователю она также будет полезна как справочное пособие. Во-первых, он найдет в ней сведения по последней версии Delphi — Delphi 5, в которой появилось много нового. А во-вторых, жизнь показывает, что обычно даже опытный разработчик предпочитает работать с очень ограниченным набором когда-то освоенных им компонентов и не использует в полной мере всего богатства библиотеки Delphi.

Книга написана так, что ее отдельные главы и разделы можно читать в произвольном порядке. Вероятно, наиболее разумно сначала проглядеть ее всю, не вникая в детали, посмотреть краткие характеристики отдельных групп компонентов и иллюстрирующие их рисунки, возможно, построить некоторые описанные в книге тестовые приложения, чтобы воочию увидеть функционирование компонентов. А потом по мере работы с Delphi 5, когда возникают те или иные задачи и проблемы, обращаться к соответствующим главам и разделам как к справочному пособию за более детальной информацией.

У читателя, знакомого с моими большим (и, увы, дорогими) книгами по аналогичной тематике, например, с книгой «Программирование в Delphi 4», может возникнуть законный вопрос: «Как соотносится материал той книги и этой, не является ли данная книга просто фрагментом большой?». Смею заверить, что не является. Конечно, часть материала повторяется. Но, прежде всего, книга посвящена Delphi 5 и в нее введен соответствующий материал по этой новой версии Delphi. Да и по традиционным компонентам материал заметно расширен по сравнению с соответствующими разделами книги «Программирование в Delphi 4». В книгу добавлено описание ряда компонентов (компонентов ActiveX, фреймов, Application, ApplicationEvents), многие компоненты (например, списки) рассмотрены существенно более детально, введено много новых примеров. Так что данную книгу можно рассматривать как значительно расширенное и переработанное издание соответствующих разделов прежней книги.

В этой небольшой вступительной главе излагаются краткие сведения о работе со средой разработки Delphi 5 и о структуре объектно-ориентированных прикладных программ. Те, кто уже работал с Delphi, могут со спокойной совестью пропустить эту главу. А тем, кто только начинает знакомство с Delphi 5, эту главу обязательно надо внимательно прочитать, чтобы спокойно воспринимать используемую далее терминологию и не испытывать трудностей с построением приложений, тестирующих различные компоненты. Для более глубокого освоения интегрированной среды разработки я рекомендую обеим категориям читателей приобрести первую книгу серии «Все о Delphi» — «Интегрированная среда разработки Delphi», в которой приводятся исчерпывающие сведения, необходимые для серьезного и продуктивного ее применения, и кроме того содержится краткий начальный курс обучения.

1.1 Интегрированная среда разработки Delphi 5

Интегрированная Среда Разработки (Integrated Development Environment — IDE, в дальнейшем мы будем использовать для нее аббревиатуру ИСР) — это среда, в которой есть все необходимое для проектирования, запуска и тестирования приложений и где все нацелено на облегчение процесса создания программ. ИСР интегрирует в себе редактор кодов, отладчик, инструментальные панели, редактор изображений, инструментарий баз данных — все, с чем приходится работать.

Запустите Delphi с помощью меню Windows Пуск | Программы. Когда вы щелкнете на пиктограмме Delphi, перед вами откроется основное окно Интегрированной Среды Разработки (см. рис. 1.1). Для версий Delphi, отличных от Delphi 5, окно выглядит несколько иначе, но в основных чертах окна ИСР всех версий Delphi одинаковы.

Подробные сведения о всех разделах меню не входят в задачу данной книги. Этому посвящена отдельная книга серии «Все о Delphi». Отметим только несколько основных команд, которые используются при разработке прикладных программ в среде Delphi (в дальнейшем изложении для краткости будем называть прикладные программы просто *приложениями*). Создание нового проекта приложения начинается с команды File | New Application. По этой команде открывается новый проект приложения с пустой формой, как вы можете это видеть на рис. 1.1. Сохранить на диске готовый проект или его заготовку можно командой File | Save Project As или File | Save All. Удобно также для сохранения использовать быстрые кнопки — третью или четвертую слева в верхнем ряду на рис. 1.1. Открыть ранее сохраненный проект можно командой File | Open или File | Open Project (вторая слева быстрая кнопка в верхнем ряду на рис. 1.1). Но если вы недавно работали с этим проектом, то удобнее воспользоваться командой File | Reopen или кнопкой справа от быстрой кнопки Open (см. рис. 1.1). Эта команда дает вам возможность быстро выбрать проект из числа тех, с которыми вы работали последнее время.

Для компиляции и запуска на выполнение приложения надо выполнить команду Run | Run (быстрая кнопка с зеленой стрелкой, пятая в нижнем ряду на рис. 1.1).

Ограничимся описанием только этих основных команд. Подробное описание всех разделов меню вы можете найти во встроенной справке Delphi, которая вызывается или из меню Help, или нажатием клавиши F1.

Палитра компонентов — витрина библиотеки визуальных компонентов (Visual Component Library — VCL). Подробное описание палитры и методики работы с ней будет дано в главе 2, а компонентам, отображаемым в палитре, посвящена вся книга. Поэтому пока ограничимся только кратким рассмотрением техники переноса компонентов из палитры на форму. Чтобы осуществить эту операцию, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. Затем надо сделать щелчок мышью в нужном месте формы и компонент разместится там. В дальнейшем вы можете отбуксировать его в любое другое место.

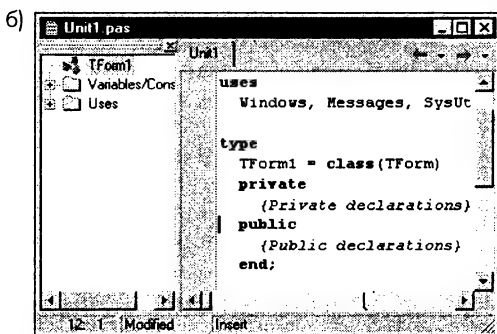
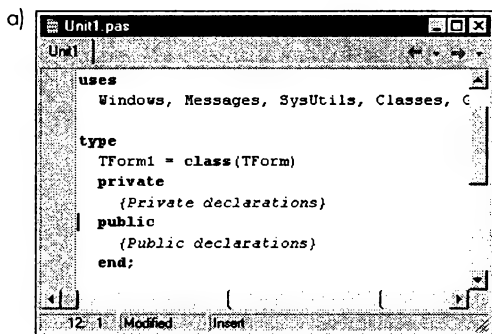
Форма, на которой размещаются компоненты, является основной почти всех приложений Delphi. Ее можно понимать как типичное окно Windows. Она имеет те же свойства, что присущи другим окнам Windows 95/98: управляющее меню в верхнем левом углу, полосу заголовка, занимающую верхнюю часть, кнопки развертывания, свертывания и закрытия окна в верхнем пра-

вом углу. Форма является контейнером (*родителем* — *Parent*) размещенных на ней компонентов. Впрочем, компоненты могут размещаться не непосредственно на форме, а в других компонентах — панелях. Тогда родителем по отношению к этим компонентам выступает соответствующая панель. Понятие родительского компонента в дальнейшем не раз будет использоваться в этой книге.

Одной из наиболее важных частей среды Delphi является окно Редактора Кода. Его вы можете видеть на рис. 1.1, а в укрупненном виде оно показано на рис. 1.2 а. В действительности, если вы откроете в первый раз это окно в Delphi 5, оно может выглядеть несколько иначе (рис. 1.2 б) и включать в себя слева еще одно встроенное окно — окно Исследователя Кода Code Explorer. Это окно в данной книге мы рассматривать не будем. В большинстве случаев вы просто можете закрыть его, щелкнув на кнопке с крестиком в его правом верхнем углу. Чтобы оно вообще больше не появлялось, выполните команду Tools | Environment Options и в открывшемся диалоговом окне на странице Explorer выключите опцию Automatically show Explorer (показывать автоматически окно Code Explorer).

Рис. 1.2

Окно Редактора Кода без встроенного окна Code Explorer (а) и со встроенным окном (б)



Редактор Кода является полноценным программным редактором. Его можно настраивать на различный стиль работы, который вам более привычен. В редакторе применяется выделение цветом синтаксических элементов. Жирным шрифтом выделяются ключевые слова языка Object Pascal (на рис. 1.2 вы видите выделение таких слов, как **type**, **class** и др.). Синим курсивом выделяются комментарии (например, на рис. 1.2 это комментарий «{Private declaration}»).

В заголовке окна Редактора Кода отображается имя текущего файла, того, с текстом которого вы работаете. В верхней части окна вы можете видеть также закладки или ярлычки, указывающие текущую страницу. Приложения Delphi могут использовать много исходных файлов и закладки помогают вам переходить от одного из них к другому.

В окно Редактора Кода, как и в другие окна Delphi, встроена контекстная справка. Чтобы получить справку по какому-то слову кода (ключевому слову, написанному имени функции и т.п.) достаточно установить курсор на это слово и нажать клавишу F1. Вам будет показана соответствующая тема справки.

Следующим важнейшим элементом среды разработки является Инспектор Объектов (Object Inspector) — см. левое окно на рис. 1.1. Он обеспечивает простой и удобный интерфейс для изменения свойств объектов Delphi и управления событиями, на которые реагирует объект. Но прежде, чем рассказывать о нем, надо сказать несколько слов о принципах объектно-ориентированного программирования, на которых основан современный подход к созданию прикладных программ.

В современном представлении программа — это система *объектов*. Каждый объект характеризуется набором *свойств*. Свойство — это некоторые данные плюс процедуры их чтения и записи в объект. Эти процедуры называются *методами* и часто работают незримо для пользователя. Пусть некоторый объект имеет свойство **A** и вы пишете в программе оператор **A := A * 10** (символы «:=» используются в языке Object Pascal для присваивания переменной, указанной в левой части оператора, значения, равного выражению в правой части). В действительности в этом случае программа вызывает метод чтения значения **A**, умножает это значение на 10, а затем вызывает метод записи значения в **A**, и передает в него вычисленную величину **A * 10**. Эти методы невидимы для пользователя. Но помимо них каждый объект обладает еще рядом методов — функций и процедур, оперирующих со свойствами объекта. Так что более полно объект можно характеризовать как совокупность свойств и методов. Перенеся объект

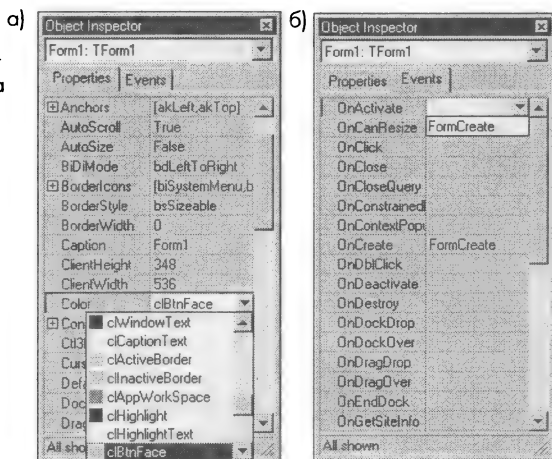
на форму и нажав клавишу F1 вы можете увидеть во встроенной справке Delphi все его свойства (properties) и методы (methods).

Помимо методов и свойств любой компонент характеризуется набором *событий*, на которое он может реагировать. Под событиями прежде всего понимаются действия пользователя: щелчок мыши, перемещение курсора, нажатие кнопок мыши или клавиш. Но и сами объекты тоже могут генерировать различные события. В объекте могут предусматриваться обработчики тех или иных событий, воспринимаемых данным компонентом. Фактически к написанию этих обработчиков и сводится программирование приложений. В обработчиках программист описывает, как должны реагировать компоненты на соответствующие события.

Завершив на этом краткий экскурс в объектно-ориентированное программирование вернемся к рассмотрению Инспектора Объектов. Окно Инспектора Объектов (рис. 1.3) имеет две страницы.

Рис. 1.3

Страница свойств (а) и страница событий (б) Инспектора Объектов



Страница свойств (Properties) Инспектора Объектов (см. рис. 1.3 а), показывает свойства того объекта, который в данный момент выделен вами. Щелкните на окне пустой формы и на странице свойств Инспектора Объектов вы сможете увидеть свойства формы (они показаны на рис. 1.3 а). Вы можете изменять эти свойства. Например, измените свойство **Caption** (надпись) вашей формы, написав в нем «Моя форма», и вы увидите, что эта надпись появится в полосе заголовка вашей формы.

Если щелкнуть на некоторых свойствах, например, на свойстве **Color** (цвет), то справа от имени свойства откроется окно выпа-

дающего списка. Нажав в нем на кнопочку со стрелкой вниз, вы можете увидеть список возможных значений свойства (см. рис. 1.3 а). Например, смените значение свойства **Color** с принятого по умолчанию **clBtnFace** (цвет поверхности кнопок) на **clWindow** (цвет окна). Вы увидите, что поверхность формы изменит свой цвет.

Рядом с некоторыми свойствами вы можете видеть знак плюс (см., например, свойство **BorderIcons** на рис. 1.3 а). Это означает, что данное свойство является объектом, который в свою очередь имеет ряд свойств.

Найдите, например, свойство **Font** (шрифт). Рядом с ним вы увидите знак плюс. Щелкните на этом плюсе или сделайте двойной щелчок на свойстве **Font**. Вы увидите, что откроется таблица таких свойств, как **Color** (цвет), **Height** (высота), **Name** (имя шрифта) и др. Среди них вы увидите свойство **Style** (стиль), около которого тоже имеется знак плюс. Щелчок на этом плюсе или двойной щелчок на этом свойстве раскроет дополнительный список подсвойств, в котором вы можете, например, установить в **true** свойство **fsBold** (жирный). Кстати, для смены **true** на **false** и обратно в подобных булевых свойствах не обязательно выбирать значение из выпадающего списка. Достаточно сделать двойной щелчок на значении свойства, и оно изменится. После того, как вы просмотрели или изменили подсвойства, вы можете опять сделать двойной щелчок на головном свойстве или щелчок на знаке минус около него, и список подсвойств свернется.

Страница событий (Events) составляет вторую часть Инспектора Объектов (см. рис. 1.3 б). На ней указаны все события, на которые может реагировать выбранный объект. Например, если вам надо выполнить какие-то действия в момент создания формы (обычно это различные операции настройки), то вы должны выделить событие **OnCreate**. Рядом с именем этого события откроется окно с выпадающим списком. Если вы уже написали в своем приложении какие-то обработчики событий и хотите при событии **OnCreate** использовать один из них, вы можете выбрать необходимый обработчик из выпадающего списка. Если же вам надо написать новый обработчик, то сделайте двойной щелчок на пустом окне списка.

Вы попадете в окно Редактора Кода, в котором увидите текст:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  
end;
```

Курсор будет расположен в пустой строке между ключевыми словами **begin** и **end**. Увиденный вами код — это заготовка обра-

ботчика события, которую автоматически сделала Delphi. Вам остается только в промежутке между **begin** и **end** написать необходимые операторы.

Если вы сделали эти операции, то вернитесь в Инспектор Объектов, выделите в нем, например, событие **OnActivate** и нажмите в нем кнопку выпадающего списка. Вы увидите в нем введенный вами ранее обработчик события **OnCreate** (этот момент изображен на рис. 1.3.6). Если вам надо использовать тот же самый обработчик и в событии **OnActivate**, просто выберите его из списка. Таким образом вы можете избежать дублирования в программе одних и тех же фрагментов кода.

Пользуясь Инспектором Объектов вы можете получить контекстную справку по свойствам или событиям. Для этого выделите в окне Инспектора Объектов интересующее вас свойство или событие и нажмите клавишу F1.

В Delphi 5 в Инспектор Объектов введена возможность фильтрации свойств и событий и возможность группировать их по категориям. Для того, чтобы воспользоваться этими возможностями, щелкните в окне Инспектора Объектов правой кнопкой мыши. Во всплывшем меню вы можете выбрать раздел **View**. Вам будет показан ряд категорий свойств и событий, около каждой из которых имеется индикатор. Вы можете включить индикаторы только у некоторых категорий и тогда в Инспекторе Объектов вы увидите события и свойства только указанных категорий. Выбор раздела **Toggle** переключит видимость разделов: те, которые были видимы станут невидимы и наоборот.

В том же меню, всплывающем при щелчке правой кнопкой мыши, вы можете выбрать раздел **Arrange** и в нем установить одну из двух возможностей: **by Name** — упорядочить свойства и события по алфавитной последовательности их имен, или **by Category** — упорядочить их по категориям. При этом некоторые свойства могут попасть одновременно в несколько категорий. Но это не имеет значения: вы можете менять их значения в любой категории и они синхронно изменятся во всех категориях.

1.2 Структура кода модуля

Рассмотрим теперь, как выглядят тексты разрабатываемых прикладных программ. Создайте простенькое приложение, в котором при щелчке пользователя на кнопке в окне появлялась бы какая-нибудь надпись. Выполните для этого последовательно следующие шаги.

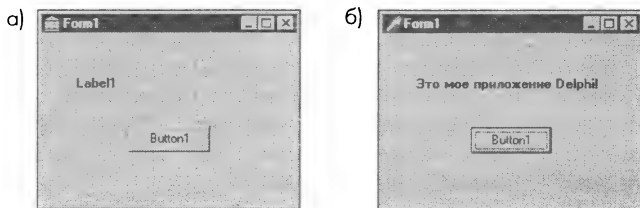
1. Запустите Delphi, если вы еще это не сделали, с помощью меню Windows Пуск | Программы. Если Delphi уже работает и вы уже делали какие-то эксперименты с формой, то откройте новое приложение. Для этого вам надо выполнить команду File | New Application. Ответьте «No» на вопрос Delphi, хотите ли вы сохранить изменения в вашем проекте
2. Перенесите на пустую форму, которая открылась вам, кнопку **Button** со страницы Standard палитры компонентов. Для этого выделите пиктограмму кнопки (она шестая слева) и затем щелкните курсором мыши в нужном вам месте формы. На форме появится кнопка, которой Delphi присвоит имя по умолчанию — **Button1**.
3. Аналогичным образом перенесите на форму с той же страницы Standard палитры компонентов метку **Label** (она на странице третья слева). В этой метке в процессе выполнения приложения будет появляться текст при нажатии пользователем кнопки. Delphi присвоит ей имя **Label1**.
4. Разместите компоненты на форме примерно так, как показано на рис. 1.4 а.
5. Выделите на форме компонент **Button1** — кнопку. Перейдите в Инспектор Объектов и измените ее свойство **Caption** (надпись), которое по умолчанию равно **Button1** (имя, которое по умолчанию присвоила этому компоненту Delphi) на «Пуск».
6. Сотрите текст в свойстве **Caption** метки **Label1**, чтобы он не высвечивался, пока пользователь не нажмет кнопку приложения.

Теперь вам осталось только написать оператор, который заносил бы в свойство **Caption** метки **Label1** нужный вам текст в нужный момент. Этот момент определяется щелчком пользователя на кнопке. При щелчке в кнопке генерируется событие **OnClick**. Следовательно, обработчик этого события вы и должны написать.

7. Выделите кнопку **Button1** на форме, перейдите в Инспектор Объектов, откройте в нем страницу событий (Events), найдите

Рис. 1.4

Форма (а) и окно в процессе выполнения (б) вашего приложения



событие кнопки **OnClick** (оно первое сверху) и сделайте двойной щелчок в окне справа от имени этого события. Это стандартный способ задания обработчиков любых событий. Вы окажетесь в окне Редактора Кода и увидите там текст:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin
```

```
end;
```

Заголовок этой процедуры складывается из имени класса вашей формы (**TForm1**), имени компонента (**Button1**) и имени события без префикса **On (Click)**.

8. Напишите в обработчике оператор задания надписи метки **Label1**. Этот оператор может иметь вид:

```
Label1.Caption:='Это мое приложение Delphi!';
```

Оператор, который вы написали, присваивает метке строку текста «Это мое приложение Delphi!».

Если вы работаете с версией старше Delphi 3, написали первый идентификатор оператора — **Label1**, поставили точку и ненадолго задумались, то вам всплывет подсказка, содержащая список всех свойств и методов метки. Это начал работать Знатор Кода — Code Insight, который стремится подсказать вам свойства и методы компонентов, аргументы функций и их типы, конструкции операторов. Вы можете выбрать из списка нужное ключевое слово, нажать клавишу Enter и выбранное слово (свойство, метод) окажется вписанным в текст.

Этот инструмент очень удобен во многих случаях, когда вы не очень точно помните последовательность перечисления параметров какой-нибудь функции, или когда не уверены в имени какого-то свойства. Но иногда, когда вы хорошо освоитесь с Delphi, эти подсказки, может быть, начнут вас раздражать. Тогда вы можете отключить Знатока Кода, выполнив команду Tools | Editor Options и на странице Code Insight выключив опцию Code Completion. В Delphi 4 аналогичные действия можно сделать командой Tools | Environment Options.

Итак, ваше приложение готово. Можете откомпилировать и выполнить его (команда Run | Run). После недолгой компиляции перед вами появится окно вашего приложения. Нажав в нем кнопку Пуск, вы увидите указанную вами строку текста (рис. 1.4 б). Можете попробовать различные манипуляции с окном: перемещение его, изменение размеров его рамки курсором мыши, свертывание и разворачивание. В заключение закройте приложение, щелкнув на кнопке в его правом верхнем углу.

Теперь посмотрите код программы, который создала с вашей помощью Delphi. Этот код содержится в двух файлах — головном файле программы с расширением **.dpr** и в файле модуля с расширением **.pas**. Головной файл **.dpr** — один на весь проект, а файлов модулей столько, сколько форм вы ввели в свой проект. Добавление в проект новой формы вы можете осуществить командой File | New Form, а удалить форму из проекта можно командой Project | Remove from Project.

Текст головного файла программы вы, как правило, не видите и не трогаете, полностью доверяя его ведение Delphi. А честь написания текстов модулей вы делите с Delphi. Delphi создает заготовку модуля, включает в нее описание формы и всех компонентов, которые вы размещаете на форме, обеспечивает связь с библиотекой. А вы пишете в основном обработчики событий компонентов. При необходимости вводите также в текст различные переменные, константы, описания типов, вспомогательные функции и процедуры. Поэтому надо представлять, как выглядит текст модуля и в какие его места что можно добавлять.

Ниже приведен текст модуля описанного выше простого примера, который вы можете увидеть в окне Редактора Кода. Подробные комментарии в этом тексте поясняют, куда и что в этот код вы можете добавлять.

```
unit Unit1;
```

```
interface // Открытый интерфейс модуля
```

```
{Список подключаемых модулей}
```

```
uses
```

```
  Windows, Messages, SysUtils, Classes, Graphics,  
  Controls, Forms, Dialogs, StdCtrls;
```

```
{Объявление класса формы}
```

```
type
```

```
  TForm1 = class(TForm)
```

```
    Label1: TLabel;
```

```
    Button1: TButton;
```

```
    procedure Button1Click(Sender: TObject);
```

```
  private // Закрытый раздел класса
```

```
    { Private declarations }
```

{Сюда могут помещаться объявления переменных, функций и процедур, включаемых в класс формы, но не доступных для других модулей}

```
  public // Открытый раздел класса
```

```
    { Public declarations }
```

{Здесь могут помещаться объявления переменных, функций и процедур, включаемых в класс формы и доступных для других модулей}

end;

var

Form1: TForm1;

{Здесь могут помещаться объявления типов, констант, переменных, функций и процедур, к которым будет доступ из других модулей, но которые не включаются в класс формы}

implementation // Реализация модуля

{\$R *.DFM}

{Здесь могут помещаться предложения uses, объявления типов, констант, переменных, к которым не будет доступа из других модулей. Тут же должны быть реализации всех объявленных в разделе interface функций и процедур, а также могут быть реализации любых дополнительных, не объявленных ранее функций и процедур.}

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Label1.Caption:='ubT XTD VxPNTFDRPD Delphi!';
end;
```

end.

Модуль начинается с ключевого слова **unit**, после которого пишется имя модуля. Оно совпадает с именем файла, в котором вы сохранили свой модуль. По умолчанию для первого модуля имя равно **Unit1**, для второго **Unit2** — и т.д.

Текст модуля состоит из двух основных разделов: **interface** — *открытый интерфейс* модуля, и **implementation** — *реализация* модуля. Все, что помещается непосредственно в раздел **interface** (типы, переменные, константы, функции, процедуры), может быть использовано другими модулями программы. Все, что помещается в раздел **implementation** — внутреннее дело модуля. Внешние модули не могут видеть типы, переменные, константы, функции и процедуры, размещенные в разделе реализации.

В разделе **interface** после предложения **uses**, содержащего список подключаемых библиотечных модулей, вы можете видеть объявление класса вашей формы, подготовленное Delphi. Имя класса вашей формы — **TForm1**. В класс включены те объекты,

которые вы разместили на форме — метка **Label1** и кнопка **Button1**. Кроме того в класс включено объявление того обработчика щелчка на кнопке, который вы создали — процедуры **Button1Click**.

В классе предусмотрено также два раздела: **private** — *закрывающийся* раздел класса, и **public** — *открытый* раздел класса. То, что вы или Delphi объявите в разделе **public**, будет доступно для других классов и модулей. То, что объявлено в разделе **private**, доступно только в пределах данного модуля.

После завершения объявления класса формы вы можете видеть строки

```
var  
  Form1: TForm1;
```

Это объявляется переменная **Form1** класса **TForm1**, т.е. объявляется ваша форма как объекта класса **TForm1**.

Затем следует раздел реализации **implementation**, который начинается с директивы компилятора, смысл которой, вообще говоря, знать не обязательно. Надо только следить, чтобы случайно не стереть эту директиву. Далее в разделе **implementation** вы можете видеть реализацию объявленной в классе процедуры **Button1Click** с единственным оператором, который вы ввели в программу сами.

В раздел **implementation** вы можете сами включать помимо обработчиков событий любые объявления глобальных переменных, констант, типов, и описания функций и процедур. Если вы хотите из данного модуля получить доступ к другому разработанному вами модулю (другой форме), то в раздел **implementation** вам надо включить оператор **uses**, в котором указать файл модуля, доступ к которому вы хотите получить. Например, предложение

```
uses Unit2, Unit3;
```

обеспечит вам доступ ко всем элементам, объявленным в интерфейсе модулей **Unit2** и **Unit3**.

1.3. Программный доступ к свойствам и методам объектов

Рассмотрим теперь, как получить из программы доступ к свойствам и методам объектов. Если вас интересует какое-то свойство объекта, то ссылка на него осуществляется в следующем формате:

```
<имя объекта>.<имя свойства>
```

После имени объекта пишется без пробела символ точки, а затем так же без пробела пишется имя свойства. Например, ссылка на свойство **Caption** метки **Label1** осуществляется записью **Label1.Caption** (см. в тексте примера предыдущего раздела процедуру **TForm1.Button1Click**).

Иногда свойство объекта является в свою очередь объектом. Тогда в обращении к этому свойству указывается через точки вся цепочка предшествующих объектов. Например, метки имеют свойство **Font** — шрифт, которое в свою очередь является объектом. У этого объекта имеется множество свойств, в частности, свойство **Color** — цвет шрифта. Чтобы сослаться на цвет шрифта метки **Label1**, надо написать **Label1.Font.Color**. Это означает: свойство **Color** объекта **Font**, принадлежащего объекту **Label1**. Например, оператор

```
Label1.Font.Color:=clRed;
```

сделает надпись метки **Label1** красной.

Аналогичная нотация с точкой используется и для доступа к методам объекта. Например, для метки, как и для большинства других объектов, определен метод **Free**, который уничтожает метку и освобождает занимаемую ею память. Если вы в какой-то момент решили уничтожить метку **Label1** в своем приложении, то вы можете написать оператор

```
Label1.Free;
```

Теперь посмотрим, чем различаются константы, переменные, функции и процедуры, включенные и не включенные в описание класса.

Если в приложении создается только один объект данного класса (в нашем примере — только один объект формы класса **TForm1**), то различие в основном чисто внешнее. Для процедур, объявленных в классе, в их описании в разделе **implementation** к имени процедуры должна добавляться ссылка на класс. В нашем примере имя процедуры **Button1Click** в описании этой процедуры в разделе **implementation** заменяется на **TForm1.Button1Click**. Для процедур, объявленных вне класса, такой замены не требуется.

Обращение к переменным и процедурам, описанным внутри и вне класса, из процедур, описанных вне класса, различается. К переменным и процедурам, описанным вне класса, обращение происходит просто по их именам, а к переменным и процедурам, описанным в классе, через имя объекта класса. Например, если вы вне класса хотите описать какую-то процедуру, изменяющую надпись метки **Label1**, вы должны обратиться к ее свойству **Caption** следующим образом: **Form1.Label1.Caption**. Только через

ссылку на объект **Form1** внешние по отношению к классу процедуры могут получить доступ ко всему, объявленному в классе.

Все эти ссылки на объект не требуются в процедурах, объявленных в классе. Поэтому в процедуре **TForm1.Button1Click** ссылка на объект **Label1** не содержит дополнительной ссылки на объект формы.

Если в приложении создается несколько объектов одного класса, например, несколько форм класса **TForm1**, то проявляются более принципиальные различия между переменными, описанными внутри и вне класса. Переменные вне класса так и остаются в одном экземпляре. А переменные, описанные в классе, тиражируются столько раз, сколько объектов данного класса создано. Т.е. в каждом объекте класса **TForm1** будут свои переменные, описанные в классе, и их значения никак не будут связаны друг с другом. Таким образом, в переменную, описанную внутри класса, можно заносить какую-то информацию, индивидуальную для каждого объекта данного класса. А переменная, описанная в модуле вне описания класса, может хранить только одно значение,

Теперь остановимся на вопросе об областях видимости элементов программы — констант, переменных, функций и процедур, т.е. о связи места их объявления в программе и места их использования. Частично этот вопрос мы уже затрагивали в предыдущем разделе, не упоминая о самом понятии *область видимости*.

Видимость отдельных элементов, размещенных в различных частях модуля, поясняется подробными комментариями в тексте модуля, приведенном в разделе 1.2. Поэтому ограничимся только некоторым подведением итогов.

- Элементы, объявленные в разделе **interface** модуля вне описания типа видимы и доступны внутри данного модуля и из внешних модулей.
- Элементы, объявленные в разделе **implementation** модуля видимы и доступны внутри данного модуля, но не доступны из внешних модулей.
- Элементы, объявленные в классе в разделе **private**, видимы и доступны только внутри данного модуля. При этом из процедур, объявленных внутри класса, к ним можно обращаться непосредственно по имени, а из других процедур — только со ссылкой на объект данного класса. Если в модуле описано несколько классов, то объекты этих классов взаимно видят элементы, описанные в их разделах **private**.
- Элементы, объявленные в классе в разделе **public**, видимы и доступны для объектов любых классов и для других модулей.

При этом из объектов того же класса к ним можно обращаться непосредственно по имени, а из других объектов и процедур — только со ссылкой на объект данного класса.

- В классах, помимо обсуждавшихся ранее, могут быть еще разделы **protected** — защищенные. Элементы, объявленные в классе в разделе **protected**, видимы и доступны для любых объектов внутри данного модуля, а также для объектов классов — наследников данного класса в других модулях. Объекты из других модулей, классы которых не являются наследниками данного класса, защищенных элементов не видят.
- Элементы, объявленные внутри другой процедуры, являются локальными, т.е. они видимы и доступны только внутри данной процедуры или внутри процедур, вложенных в данную. При этом время жизни переменных, объявленных внутри процедуры, определяется временем выполнения данной процедуры, т.е. эти переменные создаются в момент вызова процедуры и уничтожаются при завершении работы этой процедуры.

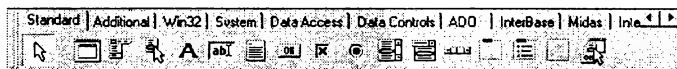
2.1 Организация библиотеки компонентов

Библиотека визуальных компонентов (Visual Component Library — VCL) Delphi содержит множество predefined типов компонентов, из которых пользователь может строить свою прикладную программу. Витрину библиотеки — палитру компонентов, вы видите расположенной справа в полосе инструментальных панелей интегрированной среды разработки Delphi. На этой палитре вы можете выделить курсором мыши нужный вам компонент и перенести его на форму. Палитра библиотеки в Delphi 5 приведена на рис. 2.1.

Рис. 2.1

Палитра

компонентов



Поскольку число страниц в палитре велико и не все закладки видны на экране одновременно, в правой части палитры компонентов имеются две кнопки со стрелками, направленными влево и вправо. Эти кнопки позволяют перемещать отображаемую на экране часть палитры.

Чтобы перенести компонент на форму, надо открыть соответствующую страницу библиотеки и указать курсором мыши необходимый компонент. При этом кнопка-указатель, размещенная в левой части палитры компонентов, приобретет вид не нажатой кнопки. Это значит, что вы находитесь в состоянии, когда собираетесь поместить компонент на форму. Поместить выбранный компонент на форму очень просто — надо сделать щелчок мышью в нужном месте формы.

Есть и другой способ поместить компонент на форму — достаточно сделать двойной щелчок на пиктограмме компонента, и он автоматически разместится в центре вашей формы. Если вы выбрали компонент, а затем изменили ваше намерение размещать его, вам достаточно нажать кнопку указателя. Это прервет процесс размещения компонента и программа вернется в нормальный режим, в котором вы можете выбирать другой компонент или выполнять какую-то команду.

Имена компонентов, соответствующих той или иной пиктограмме, вы можете узнать из ярлычка, появляющегося, если вы

задержите над этой пиктограммой курсор мыши. Если вы выберете в палитре компонент и нажмете клавишу F1, то вам будет показана справка по типу данного компонента. Тут надо сразу сделать одно замечание. Имена на ярлычках выглядят, например, так: **MainMenu**, **Button** и т.д. Однако, в Delphi все имена классов в действительности начинаются с символа «T», например, **TMainMenu**, **TButton**. Под такими именами вы можете найти описания соответствующих компонентов во встроенной в Delphi справочной системе.

Палитра имеет ряд страниц, на которых скомпонованы пиктограммы всех компонентов, определенных в Delphi. По умолчанию в палитре Delphi 5 имеются страницы:

Standard	Стандартная, содержащая наиболее часто используемые компоненты
Additional	Дополнительная, являющаяся дополнением стандартной
Win32	32-битные компоненты в стиле Windows 95/98 и NT (только начиная с Delphi 2)
System	Системная, содержащая такие компоненты, как таймеры, плееры и ряд других
Data Access	Доступ к данным через Borland Database Engine (BDE)
Data Controls	Управление данными
ADO	Связь с базами данных через Active Data Objects (ADO) — множество компонентов ActiveX, использующих для доступа к информации баз данных Microsoft OLEDB (только начиная с Delphi 5)
Interbase	Прямая связь с Interbase, минуя Borland Database Engine (BDE) и Active Data Objects (ADO) (только начиная с Delphi 5)
Midas	Построение приложений баз данных с параллельными потоками (только в вариантах Client/Server и Enterprise и только начиная с Delphi 4)
InternetExpress	Построение приложений InternetExpress — одновременно приложений сервера Web и клиента баз данных с параллельными потоками (только начиная с Delphi 5)
Internet	Интернет, компоненты для приложений, работающих с Интернет (только начиная с Delphi 2)
FastNet	Различные протоколы доступа к Интернет (только начиная с Delphi 5)

Decision Cube	Многомерный анализ данных (только в вариантах Client/Server и Enterprise, начиная с Delphi 3)
Qreport	Быстрая подготовка отчетов
Dialogs	Диалоги, системные диалоги типа «Открыть файл» и др.
Win 3.1	Windows 3.x, компоненты в стиле Windows 3.x (оставлены для обратной совместимости, начиная с Delphi 2)
Servers	Оболочки VCL для распространенных серверов COM (только начиная с Delphi 5)

Имеются еще две страницы, содержащие примеры:

Samples	Образцы: различные интересные, но не до конца документированные компоненты
ActiveX	Примеры активных элементов ActiveX

Примеры на страницах Samples и ActiveX не документированы в Delphi и во встроенной справке сведения о них отсутствуют. Однако, исходные тексты примеров со страницы Samples имеются в каталоге ...\\Source\\Samples. Вы можете их просмотреть и понять, как построены эти примеры и как ими пользоваться.

Примеры со страницы ActiveX также не документированы. Но если вы перенесете соответствующий компонент на форму и щелкнете на нем правой кнопкой мыши, то во всплывшем меню можете выбрать команду Property и некоторые другие, которые отобразят диалоговые окна, помогающие задать необходимые свойства компонента.

Многие из компонентов страниц Samples и ActiveX надо рассматривать скорее именно как примеры создания компонентов. Их полезно изучить, но для практического использования в приложениях многие из них не очень приспособлены.

В версиях Delphi, отличных от Delphi 5, страницы палитры несколько иные. Особенно отличается от других палитра в Delphi 1. Многие компоненты, которые имеются в Delphi 1 на страницах Standard, Additional и других, перенесены в последующих версиях на страницу Win 3.1 и заменены их более совершенными аналогами (см. таблицу 2.1). Компоненты страницы Win 3.1 сохраняются в 32-разрядных версиях для обеспечения совместимости при переводе приложений из Delphi 1.0 в более поздние версии. В 32-разрядных приложениях компоненты данной страницы применять не рекомендуется.

Таблица 2.1. Соответствие компонентов страницы Win 3.1 и новых 32-разрядных компонентов

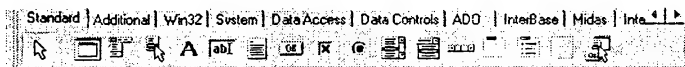
Компонент Win 3.1	Новый компонент	Страница нового компонента
DBLookupList	DBLookupListBox	Data Controls
DBLookupCombo	DBLookupComboBox	Data Controls
TabSet	TabControl	Win32
Outline	TreeView	Win32
Header	HeaderControl	Win32
NoteBook	PageControl	Win32
TabbedNoteBook	PageControl	Win32

2.2. Страницы библиотеки компонентов

В этом разделе приведены те страницы библиотеки компонентов, которые рассматриваются в данной книге. Книга посвящена компонентам общего назначения и поэтому в ней не затрагиваются страницы компонентов, используемых для работы с базами данных и для работы с Интернет. Не рассматриваются также серверы COM. Иначе говоря, из перечисленных в разделе 2.1 страниц не рассматриваются страницы Data Access, Data Controls, ADO, Interbase, Midas, InternetExpress, Internet, FastNet, Decision Cube, Servers. Рассмотрение этих страниц требует серьезного обсуждения вопросов, связанных с базами данных и Интернет, которое не укладывается в допустимый объем данной книги. По той же причине в данной книге не рассмотрены компоненты, связанные с OLE и DDE. Все эти компоненты будут рассмотрены в отдельных книгах серии «Все о Delphi», посвященных работе с базами данных и Интернет и распределенным приложениям.

2.2.1 Страница Standard

Страница Standard содержит ряд часто используемых компонентов общего назначения



Компонент	Тип	Описание
 фрейм	Frame	Панель с возможностями наследования. Проектируется как отдельное окно. Компонент визуальный.

Компонент	Тип	Описание
 главное меню	MainMenu	Позволяет конструировать и создавать полосу главного меню формы и выпадающие меню. Компонент невидимый.
 всплывающее меню	PopupMenu	Позволяет конструировать и создавать всплывающие контекстные меню, возникающие при нажатии пользователем правой кнопки мыши. Компонент невидимый.
 метка	Label	Используется для размещения на формах и других контейнерах текста, который не изменяется пользователем. Компонент визуальный.
 окно редактирования	Edit	Используется для ввода пользователем однострочных текстов. Может использоваться для отображения текста. Компонент визуальный.
 многострочное окно редактирования	Memo	Используется для ввода и отображения многострочных текстов. Компонент визуальный.
 командная кнопка	Button	Используется для создания кнопок, которыми пользователь выбирает команды в приложении. Компонент визуальный.
 контрольный индикатор с флажком	Checkbox	Позволяет пользователю включать и выключать различные опции. Компонент визуальный.
 радиокнопка	RadioButton	Предлагают пользователю набор альтернатив, из которых выбирается одна. Набор реализуется требуемым количеством радиокнопок, размещенных в одном контейнере (форме, панели и т.п.). Компонент визуальный.
 окно списка	ListBox	Представляет собой стандартное окно списка Windows, позволяющее пользователю выбирать разделы из списка. Компонент визуальный.
 редактируемый список	ComboBox	Объединяет функции ListBox и Edit. Пользователь может либо ввести текст, либо выбрать его из списка. Компонент визуальный.

Компонент	Тип	Описание
 линейка про- крутки	Scrollbar	Представляет собой стандартную линейку прокрутки Windows и служит для управления положением видимой части форм или компонентов. Компонент визуальный.
 групповое окно	GroupBox	Является контейнером, объединяющим группу связанных органов управления, таких как радиокнопки RadioButton, контрольные индикаторы Checkbox и т.д. Компонент визуальный.
 группа радио- кнопок	RadioGroup	Является комбинацией группового окна GroupBox с набором радиокнопок RadioButton; служит специально для создания групп радиокнопок. Можно размещать в компоненте несколько радиокнопок, но никакие другие органы управления не разрешены. Компонент визуальный.
 панель	Panel	Является контейнером для группирования органов управления и меньших контейнеров. Панель можно использовать также для построения полос состояния, инструментальных панелей, палитр инструментов. Компонент визуальный.
 список собы- тий	ActionList	Обеспечивает диспетчеризацию событий компонентов. Компонент невизуальный.

2.2.2 Страница Additional

Страница является дополнением страницы Standard и содержит ряд часто используемых компонентов общего назначения



Компонент	Тип	Описание
 кнопка с гра- фикой	BitBtn	Используется для создания кнопок, на которых располагается битовая графика (например, кнопка OK с галочкой). Компонент визуальный.

Компонент	Тип	Описание
 кнопка с фиксацией	SpeedButton	Используется для создания инструментальных панелей и в других случаях, когда требуется кнопка с фиксацией нажатого состояния. Компонент визуальный.
 маскированный ввод	MaskEdit	Используется для форматирования данных или для ввода символов в соответствии с шаблоном. Компонент визуальный.
 таблица строк	StringGrid	Используется для отображения текстовой информации в таблице из строк и столбцов. Компонент визуальный.
 таблица рисунков	DrawGrid	Используется для отображения в строках и столбцах нетекстовых данных. Компонент визуальный.
 изображение	Image	Используется для отображения графики: пиктограмм, битовых матриц и метафайлов. Компонент визуальный.
 формы	Shape	Используется для рисования фигур: квадратов, кругов и т.п. Компонент визуальный.
 рамка	Bevel	Используется для рисования выступающих или утопленных линий или прямоугольных рамок. Компонент визуальный.
 окно с прокруткой	ScrollBar	Используется для создания зон отображения с прокруткой. Компонент визуальный.
 список с флажками	CheckListBox	Компонент является комбинацией свойств списка ListBox и индикаторов CheckBox в одном компоненте. Компонент визуальный.
 разделитель панелей	Splitter	Используется для создания в приложении панелей с изменяемыми пользователем размерами. Компонент визуальный.
 метка с бордюром	StaticText	Компонент подобен компоненту Label, но обеспечивает дополнительные возможности по заданию стиля бордюра. Компонент визуальный.

Компонент	Тип	Описание
 инструментальная панель	ControlBar	Используется для размещения компонентов инструментальной панели. Компонент визуальный.
 события приложения	Application-Events	Перехватывает события на уровне приложения. Компонент не визуальный.
 диаграммы и графики	Chart	Компонент принадлежит к семейству компонентов TChart, которые используются для создания диаграмм и графиков. Компонент визуальный.

2.2.3 Страница Win32

Страница Win32 содержит компоненты общего назначения, позволяющие разрабатывать приложения в стиле Windows 95/98 и NT 4.x. Некоторые из этих компонентов аналогичны имеющимся на странице Win3.1.



Компонент	Тип	Описание
 страница с закладкой	TabControl	Позволяет организовывать страницы с закладками в стиле Windows 95, которые может выбирать пользователь. Компонент визуальный.
 многостраничное окно	PageControl	Позволяет создавать страницы в стиле Windows 95/98, управляемые закладками или иными органами управления, для экономии места на рабочем столе. Компонент визуальный.
 список изображений	ImageList	Предназначен для работы со списками изображений одинакового размера в меню, инструментальных панелях и т.п. Компонент не визуальный.
 окно редактирования в формате RTF	Richedit	Представляет собой окно редактирования в стиле Windows 95/98, позволяющее производить выбор цвета и шрифта, поиск текста и многое другое. Компонент визуальный.

Компонент	Тип	Описание
 ползунок	TrackBar	Управляющий элемент в виде ползунка в стиле Windows 95/98. Компонент визуальный.
 отображение хода процесса	ProgressBar	Используется для отображения в стиле Windows 95/98 хода процессов, занимающих заметное время. Компонент визуальный.
 кнопка-счетчик	UpDown	Кнопка-счетчик в стиле Windows 95/98 для ввода целых чисел. Компонент визуальный.
 «горячие» клавиши	HotKey	Дает возможность реализовать в приложении поддержку горячих клавиш. Компонент визуальный.
 воспроизведение немых клипов	Animate	Используется для воспроизведения немых клипов AVI, подобных используемым в Windows 95/98 изображениям копирования файлов и т.п. Компонент визуальный.
 ввод дат и времени	DateTime-Picker	Ввод дат и времени с выпадающим календарем. Компонент визуальный.
 ввод дат	MonthCalendar	Ввод дат с выбором из календаря. Компонент визуальный.
 дерево	TreeView	Предоставляет возможность просмотра структуры иерархических данных в стиле Windows 95/98. Компонент визуальный.
 списки	ListView	Отображает списки в стиле Windows 95/98. Компонент визуальный.
 заголовок	HeaderControl	Позволяет создавать составные перемещаемые заголовки в стиле Windows 95/98. Компонент визуальный.
 полоса состояния	StatusBar	Полоса состояния программы, при необходимости — на нескольких панелях. Компонент визуальный.
 инструментальная панель	ToolBar	Инструментальная панель для быстрого доступа к часто используемым функциям приложения. Компонент визуальный.

Компонент	Тип	Описание
 инструментальная пере- страиваемая панель	CoolBar	Контейнер инструментальной панели, размеры которой могут изменяться пользователем. Компонент визуальный.
 прокрутка страниц	PageScroller	Обеспечивает прокрутку больших окон, например, инструментальных панелей. Компонент визуальный.

2.2.4 Страница System



Страница System содержит компоненты, позволяющие использовать системные средства Windows.

Компонент	Тип	Описание
 таймер	Timer	Используется для запуска процедур, функций и событий в указанные интервалы времени. Компонент не визуальный.
 окно для ри- сования	PaintBox	Используется для создания на форме некоторой области, в которой можно рисовать. Компонент визуальный.
 аудио и видео плеер	MediaPlayer	Используется для создания панели управления воспроизведением звуковых и видео файлов, а также устройств мультимедиа. Компонент визуальный.
 контейнер OLE	OLEContainer	Используется при создании области клиента для объекта OLE. Компонент визуальный.
 диалог с сер- вером DDE	DDEClient- Conv	Используется клиентом DDE для организации диалога с сервером DDE. Компонент не визуальный.

Компонент	Тип	Описание
 данные, передаваемые серверу DDE	DDEClientItem	Используется для определения данных клиента, передаваемых в диалоге серверу DDE. Компонент невидимый.
 диалог с клиентом DDE	DDEServerConv	Компонент используется сервером DDE при проведении диалога с клиентом DDE. Компонент невидимый.
 данные, передаваемые клиенту DDE	DDEServerItem	Компонент используется для определения данных сервера, передаваемых клиенту DDE в течение диалога. Компонент невидимый.

2.2.5 Страница QReport



Страница QReport содержит компоненты, используемые при генерации отчетов.

Компонент	Тип	Описание
 отчет	QuickRep	Используется для введения в приложение средств печати отчетов QuickReport. Компонент невидимый.
 детали	QRSubDetail	Используется для компоновки в отчет дополнительных данных. Компонент визуальный.
 полоса текста	QRStringsBand	Используется для компоновки в отчет дополнительных текстов. Компонент визуальный.
 полоса	QRBand	Используется для построения отчетов путем размещения на нем печатаемых компонентов. Компонент визуальный.
 дочерняя полоса	QRChildBand	Используется для создания дочерних полос, которые могут содержать другие компоненты QuickRep и полосы. Компонент визуальный.

Компонент	Тип	Описание
 группировка	QRGroup	Используется для группировки данных. Компонент невизуальный.
 метка	QRLabel	Используется для размещения текста в отчете. Компонент визуальный.
 текст из базы данных	QRDBText	Представляет собой ориентированный на данные компонент для размещения текста в отчете. Компонент визуальный.
 математические выражения	QRExpr	Позволяет строить и отображать выражения над полями данных и системными величинами (такими, как время и дата).
 системные данные	QRSysData	Используется для отображения системных данных. Компонент визуальный.
 многострочный текст	QRMemo	Используется для размещения в отчете многострочных текстов. Компонент визуальный.
 тексты с математическими выражениями	QRExprMemo	Используется для размещения в отчете текстов с математическими выражениями. Компонент визуальный.
 многострочный текст RTF	QRRichText	Используется для размещения в отчете текста в обогащенном формате RichText . Компонент визуальный.
 многострочный текст RTF базы данных	QRDBRichText	Используется для размещения в отчете текста из базы данных в обогащенном формате RichText . Компонент визуальный.
 форма	QRShape	Используется для рисования в отчете графических форм. Компонент визуальный.

Компонент	Тип	Описание
 изображение	QRImage	Используется для печати изображений в отчете. Компонент визуальный.
 изображение из базы данных	QRDBImage	Используется для печати изображений из баз данных в отчете. Компонент визуальный.
 составной отчет	QRComposite-Report	Используется для построения составных отчетов. Компонент визуальный.
 предварительный просмотр	QRPreview	Используется для предварительного просмотра на экране подготовленного к печати отчета. Компонент визуальный.
 фильтр текста	QRTextFilter	Используется для установки фильтра текста. Компонент невидимый.
 разделитель	QRCSVFilter	Используется для установки разделителя текста. Компонент невидимый.
 фильтр HTML	QRHTMLFilter	Используется для установки фильтра HTML. Компонент невидимый.
 диаграммы, графики	QRChart	Используется для печати в отчете диаграмм, построенных на основе баз данных. Компонент визуальный.

2.2.6 Страница Dialogs

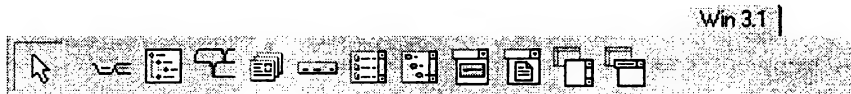
Dialogs |



Страница Dialogs содержит компоненты, используемые для создания различных диалоговых окон, общепринятых в приложениях Windows. Диалоги используются для указания файлов или выбора установок. Применение предоставляемых в составе Delphi диалоговых окон помогает сэкономить время на разработку и придать вашему приложению совместимость с принятыми в Windows нормами диалога.

Компонент	Тип	Описание
 «Открыть файл»	OpenDialog	Предназначен для создания окна диалога «Открыть файл». Компонент невидимый.
 «Сохранить файл как...»	SaveDialog	Предназначен для создания окна диалога «Сохранить файл как...». Компонент невидимый.
 «Открыть рисунок»	OpenPictureDialog	Предназначен для создания окна диалога «Открыть рисунок». Компонент невидимый.
 «Сохранить рисунок как...»	SavePictureDialog	Предназначен для создания окна диалога «Сохранить рисунок как...». Компонент невидимый.
 «Шрифты»	FontDialog	Предназначен для создания окна диалога «Шрифты». Компонент невидимый.
 «Цвет»	ColorDialog	Предназначен для создания окна диалога «Цвет». Компонент невидимый.
 «Печать»	PrintDialog	Предназначен для создания окна диалога «Печать». Компонент невидимый.
 «Установка принтера»	PrinterSetupDialog	Предназначен для создания окна диалога «Установка принтера». Компонент невидимый.
 «Найти»	FindDialog	Предназначен для создания окна диалога «Найти». Компонент невидимый.
 «Заменить»	ReplaceDialog	Предназначен для создания окна диалога «Заменить». Компонент невидимый.

2.2.7 Страница Win3.1



Страница Win3.1 содержит компоненты, которые широко используются Delphi 1, но в более поздних версиях Delphi заменены новыми. Компоненты этой страницы сохраняются в 32-разрядных версиях для обеспечения совместимости при переводе приложений из Delphi 1.0 в более поздние версии. В 32-разрядных приложениях компоненты данной страницы применять не следует.

Компонент	Тип	Описание
 блокнот с закладками	TabSet	Используется для создания блокнота с закладками. Компонент визуальный.
 окно дерева	Outline	Позволяет отображать иерархические данные в форме дерева. Компонент визуальный.
 многостраничная форма	TabbedNoteBook	Используется для создания многостраничных форм с закладками. Компонент визуальный.
 пачка страниц	NoteBook	Используется для создания пачки страниц, может применяться совместно с TabSet. Компонент визуальный.
 заголовок	Header	Используется для отображения текста в областях переменного размера. Компонент визуальный.
 список файлов	FileListBox	Отображает список файлов каталога. Компонент визуальный.
 структура каталогов	DirectoryListBox	Отображает структуру каталогов диска. Компонент визуальный.
 список дисков	DriveComboBox	Выпадающий список доступных дисков. Компонент визуальный.

Компонент	Тип	Описание
 список фильтров	FilterComboBox	Выпадающий список фильтров для поиска файлов. Компонент визуальный.
 создание списка данных	DBLookupList	Предназначен для просмотра значений в таблице данных с помощью окна списка. Компонент визуальный.
 создание редактируемого списка данных	DBLookupCombo	Предназначен для просмотра значений в таблице данных с помощью выпадающего списка. Компонент визуальный.

2.2.8 Страница Samples



Страница Samples содержит примеры компонентов. Поскольку это всего лишь примеры, они снабжены в Delphi лишь минимальной документацией и во встроенной справке сведения о них отсутствуют. Однако, исходные тексты примеров со страницы Samples имеются в каталоге ...\\Source\\Samples. Вы можете их просмотреть и понять, как построены эти примеры и как ими пользоваться.

Компонент	Тип	Описание
 индикатор хода процесса	Gauge	Пример компонента, используемого для создания индикатора хода процесса в виде линейки, текста или секторной диаграммы. Компонент визуальный.
 таблица цветов	ColorGrid	Пример компонента, используемого для создания таблицы цветов, в которой пользователь выбирает требуемый цвет. Компонент визуальный.

Компонент	Тип	Описание
 кнопка-счетчик	SpinButton	Пример компонента, используемого для создания кнопок-счетчиков. Компонент визуальный.
 окно редактирования со счетчиком	SpinEdit	Пример компонента, используемого для создания окна редактирования в комбинации с кнопкой-счетчиком. Компонент визуальный.
 дерево каталогов	DirectoryOutline	Пример компонента, используемого для отображения структуры каталогов выбранного диска. Компонент визуальный.
 календарь	Calendar	Пример компонента, используемого для отображения календаря на указанный месяц в стандартном формате. Компонент визуальный.
 индикатор события	IBEventAlerter	Пример компонента, сигнализирующего о событии в базе данных. Компонент не визуальный.

2.2.9 Страница ActiveX



Страница ActiveX содержит примеры компонентов ActiveX. Поскольку это всего лишь примеры, они снабжены в Delphi лишь минимальной документацией и во встроенной справке сведения о них отсутствуют. Но если вы перенесете соответствующий компонент на форму и щелкнете на нем правой кнопкой мыши, то во всплывшем меню можете выбрать команду **Property** и некоторые другие, которые отобразят диалоговые окна, помогающие задать необходимые свойства компонента.

Компонент	Тип	Описание
 диаграммы и графики	Chartfx	Редактор диаграмм и графиков.
 орфографиче- ский контроль	VCSpell	Визуальный блок орфографическо- го контроля.
 страницы Excel	F1Book	Компонент ввода и обработки чис- ловой информации, аналогичный страницам Excel.
 диаграммы	VtChart	Окно построения диаграмм.
 диаграммы и графики	Graph	Окно построения диаграмм и гра- фиков.

2.3. Настройка палитры компонентов

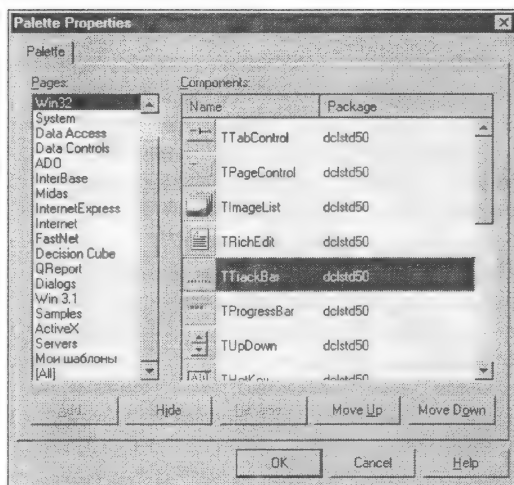
В предыдущих разделах описаны страницы палитры компонентов по умолчанию. Но палитру компонентов можно перестраивать, перемещать компоненты с одной страницы на другую, вообще убирать не используемые вами компоненты с палитры, изменять последовательность страниц, перемещая в конец палитры те страницы, к которым вы редко или вообще никогда не обращаетесь. Можно также создавать новые страницы. Можно включать на страницы свои собственные шаблоны компонентов или разработанные вами новые компоненты.

Вызвать настройку палитры компонентов можно щелчком правой кнопки мыши на палитре и выбором команды *Properties* в Delphi 5 и 4 или *Configure* в более ранних версиях из всплывшего меню. Можно выполнить для этого команду *Component | Configure Palette*. Можно также выполнить команду *Tools | Environment Options* в Delphi 3 — 5 или *Options | Environment* в Delphi 1. При таком вызове настройки надо перейти в открывшемся диалоговом окне на страницу *Palette*. Вид окна настройки палитры компонентов представлен на рис. 2.2.

Опции окна позволяют работать со страницами палитры. Для этого надо перейти в окно *Pages* и нажать кнопку *Add*, чтобы до-

Рис. 2.2

Окно настройки палитры компонентов



бавить новую страницу (см. страницу Мои шаблоны на рис. 2.2), кнопку Rename, чтобы переименовать страницу, кнопку Delete, чтобы удалить страницу (она должна быть к этому моменту пустой), кнопки Move Up или Move Down, чтобы изменить последовательность страниц в палитре.

Перейдя в окно Components вы увидите состав той страницы, которая выделена в окне Pages. Если вы в окне Pages выделили раздел [All], то в окне компонентов вы увидите все компоненты, включенные в библиотеку. Вы можете изменять состав страниц, перетаскивая мышью компонент из окна Components в окно Pages на какую-то другую страницу. Если вы перетащите компонент в раздел [All], он не будет присутствовать ни на одной странице, но останется в библиотеке и впоследствии вы сможете при желании перетащить его из раздела [All] на какую-то страницу палитры. Кнопкой Hide вы можете сделать какие-то компоненты невидимыми на странице.

2.4. Создание и запись в библиотеку шаблонов компонентов и групп компонентов

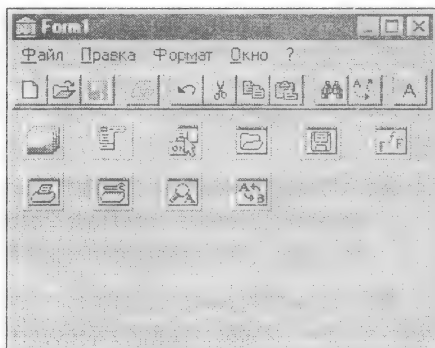
Вы можете использовать страницы библиотеки для включения в них каких-то разработанных вами шаблонов компонентов или совокупностей компонентов. В последующих главах будут рассмотрены различные компоненты. В каждом из них в процессе проектирования задается некоторое множество значений

свойств и пишутся обработчики событий. Причем очень часто компоненты с одинаковыми свойствами и с одинаковыми или почти одинаковыми обработчиками кочуют из приложения в приложение. Например, стандартные меню с одними и теми же разделами имеются в очень многих приложениях. Причем меню должны быть именно стандартными, с одними и теми же надписями в разделах, с одними и теми же «горячими» клавишами и т.п. Такая стандартизация — одно из основных требований, предъявляемых к приложениям Windows. Кроме того каждое меню по современным требованиям должно сопровождаться инструментальной панелью с быстрыми кнопками, дублирующими основные разделы меню. Эти быстрые кнопки должны иметь привычные пользователю пиктограммы, должны сопровождаться всплывающими ярлычками подсказок. Наконец, любое меню, имеющее разделы работы с файлами должно вызывать стандартные диалоги Windows открытия и закрытия файлов.

Все это легко создать в Delphi, но для этого надо перенести на форму ряд компонентов (см. пример на рис. 2.3): MainMenu — главное меню (см. раздел 6.1), инструментальную панель ToolBar (см. раздел 7.4) и создать на ней множество быстрых кнопок, различные стандартные диалоги (см. главу 7), список изображений для кнопок и разделов меню ImageList (см. раздел 9.3), диспетчер событий ActionList (см. раздел 9.2). В каждом из этих компонентов надо задать различные свойства, связать компоненты друг с другом, написать обработчики событий. Это немалая работа. И обидно, если начиная в дальнейшем похожие приложения придется выполнять ее снова и снова. Вот тут и могут помочь шаблоны, внесенные вами в библиотеку компонентов.

Рис. 2.3

Пример меню с инструментальной панелью и стандартными диалогами



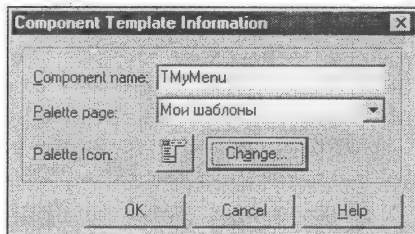
Чтобы сохранить в библиотеке как шаблон компонент или группу компонентов вместе со всеми значениями свойств компонентов и обработчиками событий, надо сделать следующее.

1. Задайте значения всех необходимых свойств в компонентах, которые вы хотите оформить как шаблон. Напишите требуемые обработчики событий.
2. Выделите на форме группу ваших компонентов. Если компоненты расположены непосредственно на форме, то для выделения их как группы достаточно построить курсором мыши рамку, охватывающую или хотя бы пересекающую все эти компоненты. Так можно выделить в группу компоненты, расположенные рядом. Если компоненты расположены не рядом друг с другом или они расположены на какой-то панели, то применима другая методика: нажмите клавишу Shift и, не отпуская ее, щелкните поочередно на каждом компоненте, включаемом в группу.
3. Выполните команду **Component | Create Component Template** (создать шаблон компонента). Перед вами откроется диалоговое окно **Component Template Information** (рис. 2.4). В его верхнем окошке редактирования **Component Name** вы можете задать имя шаблона. Это имя будет появляться на ярлычке подсказки, если пользователь задержит курсор мыши над пиктограммой вашего шаблона в палитре библиотеки. На рис. 2.4 это имя — **TMyMenu**. В выпадающем списке в средней части окна вы можете выбрать страницу библиотеки визуальных компонентов, на которой хотите разместить пиктограмму шаблона. Вы можете также указать новое имя (Мои шаблоны на рис. 2.4) и тогда в библиотеке визуальных компонентов будет создана новая страница с этим именем. Можно также изменить пиктограмму данного компонента (кнопка **Change**). Пиктограмма, если вы ее хотите сменить, должна быть подготовлена заранее в виде файла **.bmp** размером 24 на 24.
4. После выполнения всех описанных операций щелкните на кнопке **OK**.

И это все. Ваш шаблон появится в библиотеке. Вы можете убедиться в этом, посмотрев на указанную вами страницу библиотеки. Попробуйте провести подобный эксперимент с компонентами любого приложения. Например, с элементарным приложением, рассмотренным в разделе 1.2 (рис. 1.4) и содержащим всего два компонента: кнопку и метку, в которую при щелчке на кнопке пишется какой-то текст. Оформите эти компоненты как шаблон и занесите в библиотеку. Затем откройте новое приложение и пе-

Рис. 2.4

Диалоговое окно задания информации о шаблоне



перенесите на форму из библиотеки несколько экземпляров шаблона. В вашем проекте появится несколько кнопок и несколько меток, причем каждая кнопка будет управлять своей меткой. Поскольку метки будут иметь разные имена, то, посмотрев код модуля, вы увидите, что Delphi учла это: операторы обработчиков щелчков на разных кнопках различаются именами кнопок.

Таким простым способом вы можете создать себе немало шаблонов компонентов или групп компонентов, которые кочуют из приложения в приложение. Подобное повторное использование разработанных вами кодов сэкономит вам много времени при создании новых приложений.

2.5 Некоторые итоги

В данной главе мы рассмотрели работу с библиотекой компонентов. Последующие главы книги посвящены обзору различных компонентов, хранящихся в библиотеке. При этом тематика отдельных глав будет определяться не страницами палитры, а функциям, выполняемыми компонентами. Это позволит сравнивать компоненты, близкие по своему функциональному назначению, видеть их достоинства и недостатки и вырабатывать рекомендации по их применению.

3.1 Font — шрифт отображения текста в компонентах

Во всех компонентах ввода и отображения текстовой и цифровой информации шрифт текста, его размер, стиль (жирный, курсив и т.п.) определяются свойством **Font**. Это свойство в свою очередь является объектом типа **TFont**, который имеет множество подсвойств, устанавливаемых в процессе проектирования или программно во время выполнения приложения. Основные свойства шрифта следующие:

Charset	Определяет набор символов шрифта.
Color	Определяет цвет текста.
Height	Характеризует высоту шрифта в пикселях.
Name	Вид (имя) шрифта.
Pitch	Определяет способ установки ширины символов.
Size	Размер шрифта в кеглях (пунктах).
Style	Стиль шрифта: полужирный — fsBold , курсив — fsItalic , подчеркнутый — fsUnderline , перечеркнутый — fsStrikeout .

Если система не может найти шрифта с заданной комбинацией свойств **Name**, **Charset**, **Pitch** и **Size**, Windows использует другой, близкий по характеристикам шрифт.

Основное свойство шрифта — **Name**. Если заданный именем **Name** шрифт недоступен в системе, Windows заменит его другим шрифтом. По умолчанию для всех компонентов Delphi, начиная с Delphi 2, задается имя шрифта **MS Sans Serif** и размер 8. Delphi 1 задает имя шрифта **System** и размер 10. Можно рекомендовать без особой нужды не изменять **Name**, так как заданные по умолчанию шрифты есть на любом компьютере с Windows, а другой

шрифт может отсутствовать на компьютере пользователя вашей программы.

Свойства **Size** и **Height** определяют размер шрифта. Свойство **Size** определяет его в кеглях (пунктах, принятых в Windows), а свойство **Height** — в пикселях. Если значение **Size** задано отрицательным, то в размер входит верхний пиксель каждой строки. Если значение **Size** задано положительным, то этот пиксель не учитывается.

Значение **Size** связано со свойствами **Height** и **PixelsPerInch** (число пикселей на дюйм) соотношением:

$$\text{Font.Size} = -\text{Font.Height} * 72 / \text{Font.PixelsPerInch}$$

Из соотношения, в частности, видно, что задание положительного значения **Size** ведет к отрицательному значению **Height** и наоборот.

Свойство **Pitch** обычно имеет значение **fpDefault**, при котором ширина устанавливается равной по умолчанию, т.е. описанной в шрифте заданного вида **Name**. Свойство может принимать также значения **fpFixed** — установка одинаковой ширины всех символов и **fpVariable** — установка различной ширины символов. Задание значения отличного от **fpDefault** заставляет Windows искать наилучший способ удовлетворить всем заданным характеристикам шрифта. Иногда это может привести к замене шрифта на шрифт другого, близкого вида, а иногда может вообще не повлиять на шрифт — все зависит от конкретного вида шрифта и даже от его версии.

Свойство **Charset** определяет набор символов шрифта. Каждый вид шрифта, определяемый его именем, поддерживает один или более наборов символов. Какие именно значения **Charset** поддерживает тот или иной шрифт можно установить из документации на него или экспериментальным путем, в частности, с помощью приведенного далее в этом разделе тестового приложения. Для шрифтов, поддерживающих несколько наборов символов, важно правильно задать **Charset**.

В Delphi предопределено много констант, соответствующих стандартным наборам символов. Большинство из них, относящихся к японскому, корейскому, китайскому и другим языкам, вряд ли представляют интерес для наших читателей. Поэтому отметим только одно значение — 204, обозначаемое также константой **RUSSIAN_CHARSET**, которое соответствует символам кириллицы.

По умолчанию в объектах типа **TFont** задается значение **Charset**, равное 1 или **DEFAULT_CHARSET**. При этом шрифт выбирается только по его имени **Name** и размеру **Size**. Если описан-

ный шрифт недоступен в системе, то Windows заменит его другим шрифтом. Для имен шрифтов, принятых в Delphi по умолчанию, это обычно нормальный вариант. Но в ряде случаев полезно для отображения русских текстов с другими шрифтами заменить это значение на **RUSSIAN_CHARSET**. Это позволит отобразить символы кириллицы для тех шрифтов, для которых при **DEFAULT_CHARSET** символы кириллицы не отображаются нормально.

Свойство **Style**, задающее стиль, представляет собой множество или пустое, или содержащее одно или более из возможных значений. Ниже приведены примеры операторов, устанавливающих стиль шрифта:

```
Label1.Font.Style := [ ];           {Обычный стиль}
Label1.Font.Style := [fsBold];      {Полужирный}
Label1.Font.Style := [fsBold, fsItalic]; {Полужирный курсив}
```

Значения свойств объекта **Font** можно присваивать по отдельности, как это сделано в приведенных выше операторах для свойства **Style**. Но чаще они задаются все сразу методом **Assign**, который записывает значения всех свойств одного объекта в другой. Пусть, например, на форме имеется компонент **Memol** (см. раздел 3.3.4), в котором расположен некоторый текст, компонент **FontDialog1** — диалог выбора шрифта (см. раздел 8.4), и меню с разделом выбора шрифта, названным **MFont**. Для того, чтобы пользователь мог выбрать имя и атрибуты шрифта текста, отображаемого в **Memol**, в обработчик события **OnClick** раздела меню **MFont** надо вставить оператор:

```
if (FontDialog1.Execute)
  then Memol.Font.Assign(FontDialog1.Font);
```

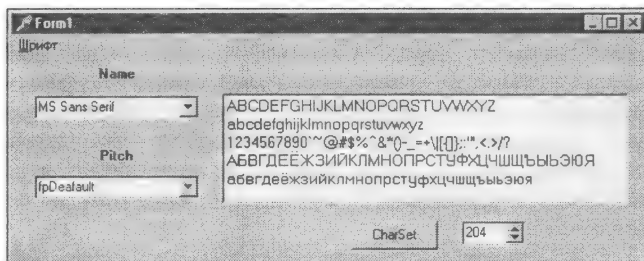
Если пользователь сменил атрибуты в диалоговом окне выбора шрифта, то метод **FontDialog1.Execute** (см. раздел 8.1) возвращает **true** и атрибуты шрифта компонента **Memol** устанавливаются равными выбранным пользователем.

Для того, чтобы продемонстрировать доступные в системе шрифты и исследовать влияние на них свойств **Pitch** и **Charset**, можно построить тестовое приложение, показанное на рис. 3.1 В нем используются компоненты **MainMenu**, **Button**, **ComboBox**, **SpinEdit** и **Memo**, которые будут рассмотрены позднее в разделах 6.1, 5.2, 3.3.5, 3.4.2 и 3.3.4. Поэтому те, кто не знаком с этими компонентами, могут пока пропустить данный пример и вернуться к нему позднее.

Разместите на форме многострочное окно редактирования **Memo** (см. раздел 3.3.4), в который поместите какой-нибудь

Рис. 3.1

Тестовое приложение свойств шрифта



текст, например, тест латинских и русских символов, показанный на рис. 3.1. Перенесите на форму выпадающий список **ComboBox** (см. раздел 3.3.5), в который будут загружаться имена шрифтов, доступных системе на данном компьютере. Назовите этот компонент **CBName**. Разместите еще один компонент **ComboBox**, который будет содержать возможные значения свойства **Pitch**. Назовите его **CBPitch** и занесите в его свойство **Items** строки «fpDefault», «fpFixed» и «fpVariable». Разместите на форме также компонент **SpinEdit** (см. раздел 3.4.2), в котором можно будет задавать численные значения свойства **CharSet**, и кнопку **Button** (см. раздел 5.2), нажатие которой будет передавать значение, введенное пользователем в **SpinEdit**, в свойство шрифта **CharSet**. Разместите на форме компонент главного меню **MainMenu** (см. раздел 6.1) и введите в меню один раздел — Шрифт, при выборе которого пользователь сможет в диалоге выбирать атрибуты шрифта, в частности, его размеры и стиль. Чтобы обеспечить диалог выбора шрифта разместите на форме также компонент **FontDialog** (см. раздел 8.4).

Расположение всех компонентов может примерно соответствовать приведенному на рис. 3.1. Далее надо написать обработчики событий: события **OnCreate** при создании формы (**FormCreate**), выбора раздела меню (**MFontClick**), изменений в списке **CBName** (**CBNameClick**) и в списке **CBPitch** (**CBPitchChange**) и щелчка на кнопке **Button1** (**Button1Click**). Ниже приведен текст всех этих обработчиков.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i: integer;
begin
  // Загрузка в CBName всех шрифтов системы
  for i := 0 to Screen.Fonts.Count - 1 do
    CBName.Items.Add(Screen.Fonts[i]);
  CBName.ItemIndex := 0;
  CBPitch.ItemIndex := 0;
  // Загрузка в Memo1 имени первого шрифта
```

```
    Mem1.Font.Name := CName.Items[CName.ItemIndex];
end;

procedure TForm1.MFontClick(Sender: TObject);
begin
    // Задание диалогу текущих атрибутов шрифта Mem1
    FontDialog1.Font.Assign(Mem1.Font);
    if (FontDialog1.Execute)
        then begin
            // Задание атрибутов шрифта, выбранных в диалоге
            // пользователем
            Mem1.Font.Assign(FontDialog1.Font);
            CName.Text:=Mem1.Font.Name;
            SpinEdit1.Value:=FontDialog1.Font.CharSet;
        end;
end;

procedure TForm1.CNameChange(Sender: TObject);
begin
    // Изменение свойства Name
    Mem1.Font.Name := CName.Items[CName.ItemIndex];
end;

procedure TForm1.CBPitchChange(Sender: TObject);
begin
    // Изменение свойства Pitch
    case CBPitch.ItemIndex of
        0: Mem1.Font.Pitch := fpDefault;
        1: Mem1.Font.Pitch := fpFixed;
        2: Mem1.Font.Pitch := fpVariable;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
    // Изменение свойства CharSet
    Mem1.Font.Charset := SpinEdit1.Value;
end;
```

Комментарии в тексте поясняют отдельные операции. Запустите приложение и исследуйте шрифты, зарегистрированные в системе, и влияние на них свойств **Pitch** и **Charset**.

3.2 Вывод текста на канву Canvas

В последующих разделах будут рассмотрены специализированные компоненты отображения и ввода текстовой и цифровой информации. Но помимо этих компонентов отображать текстовые надписи можно непосредственно на свойстве **Canvas** (канва,

холст) любого компонента, имеющего это свойство, в частности, непосредственно на фоне.

При выводе текста на канву фон надписи определяется свойством канвы **Brush** — кисть. Это свойство является в свою очередь объектом, который может хранить в своем свойстве **Style** некий шаблон заполняющей штриховки. Но по умолчанию этот шаблон соответствует сплошной закрашке фона. А свойство кисти **Color** задает цвет фона. Атрибуты выводимого текста определяются рассмотренным в разделе 3.1 свойством канвы **Font**.

Вывод текста на канву может осуществляться методом **TextOut**, объявленным следующим образом:

```
procedure TextOut(X, Y: Integer; const Text: string);
```

Процедура **TextOut** пишет строку текста **Text** на канве, начиная с позиции с координатами (**X**, **Y**). Если цвет кисти **Brush** в момент вывода текста отличается от того, которым закрашена канва, то текст получится выведенным в цветной прямоугольной рамке. Но ее размеры будут точно равны размерам надписи. Если требуется более красивая рамка с отступом от текста, следует применять другой метод — **TextRect**. Этот метод определен следующим образом:

```
procedure TextRect(Rect: TRect; X, Y: Integer;  
                  const Text: string);
```

Процедура **TextRect** пишет строку текста **Text** на канве, начиная с позиции с координатами (**X**, **Y**) — это левый верхний угол надписи. Параметр **Rect** типа **TRect** определяет прямоугольную рамку, в которую вписывается текст. Тип **TRect** определен следующим образом:

```
TRect = record  
  case Integer of  
    0: (Left, Top, Right, Bottom: Integer);  
    1: (TopLeft, BottomRight: TPoint);  
end;
```

Координаты задаются и как четыре целых числа, определяющих координаты в пикселях левой (**Left**), верхней (**Top**), правой (**Right**) и нижней (**Bottom**) сторон прямоугольника, и как две точки типа **TPoint**, представляющие собой координаты левого верхнего и правого нижнего углов:

```
type TPoint = record  
  X: Longint;  
  Y: Longint;  
end;
```

Началом координат обычно считается левый верхний угол экрана или окна.

При выводе текста методом **TextOut** часть текста, не помещающаяся в прямоугольную область **Rect**, отсекается.

Надписи и в методе **TextOut**, и в методе **TextRect** делаются в соответствии с текущими установками шрифта **Font**. Пространство внутри области **Rect** в методе **TextRect** и фон надписи в методе **TextOut** закрашиваются текущей кистью **Brush**.

Для выравнивания позиции текста на канве и для задания красивой рамки в методе **TextRect** можно использовать методы, дающие высоту и длину выводимого текста в пикселях: методы **TextWidth**, **TextHeight** и **TextExtent**. Функция **TextWidth**:

```
function TextWidth(const Text: string): Integer;
```

возвращает длину в пикселях текста **Text**, который предполагается написать на канве текущим шрифтом. Функция **TextHeight**:

```
function TextHeight(const Text: string): Integer;
```

возвращает высоту в пикселях текста **Text**. Применение методов **TextWidth** и **TextHeight** позволяет перед выводом текста на канву определить размер надписи и расположить ее и другие элементы изображения наилучшим образом.

Имеется еще метод **TextExtent**, определенный следующим образом:

```
type
  TSize = record
    cx: Longint;
    cy: Longint;
  end;
```

```
function TextExtent(const Text: string): TSize;
```

Метод возвращает одновременно и высоту, и длину текста. Метод **TextWidth** возвращает то же, что **TextExtent(Text).cx**, а метод **TextHeight** возвращает то же, что **TextExtent(Text).cy**.

Рассмотрим примеры применения всего этого на практике. Оператор

```
Form1.Canvas.TextOut(10,10,s);
```

выводит текст, хранящийся в строковой переменной **s**, на канву формы **Form1**, начиная с позиции (10, 10). Если форма **Form1** является той, в обработчике которой написан этот оператор, то ссылку на **Form1** можно не делать:

```
Canvas.TextOut(10,10,s);
```

Оператор

```
with Canvas do
  TextOut((ClientWidth - TextWidth(s)) div 2,
    TextHeight(s),s);
```

выводит текст на канву текущей формы, выравнивая его при любом шрифте по середине ширины канвы (ширина определяется свойством **ClientWidth**) и отступив одну строчку сверху. То же самое делает и оператор

```
with Canvas do
  TextOut((ClientWidth - TextExtent(s).cx) div 2,
          TextExtent(s).cy, s);
```

который вместо методов **TextWidth** и **TextHeight** использует метод **TextExtent**.

Оператор

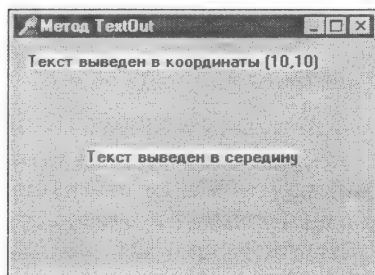
```
with Canvas do
  TextOut((ClientWidth - TextWidth(s)) div 2,
          (ClientHeight - TextHeight(s)) div 2, s);
```

выводит текст в середину формы и по высоте и по ширине.

Примеры вывода текста методом **TextOut** приведены на рис. 3.2. Основной недостаток такого вывода заключается в том, что рамка текста получается без зазоров. Такой вывод выглядит некрасиво. Вероятно, вывод этой функцией оправдан только в случае, если цвет фона (свойство **Canvas.Brush.Color**) совпадает с цветом поверхности компонента, на канву которого выводится текст. В этом случае он просто появится без рамки.

Рис. 3.2

Примеры вывода текста методом **TextOut**



Более изящный вывод обеспечивает метод **TextRect**. Следующий код рисует в заданном месте канвы формы с координатами (**X1,Y1,X2,Y2**) красный прямоугольник и внутри него в центре пишет методом **TextRect** текст, введенный в переменную **S**. Если текст оказывается длиннее ширины прямоугольника, то он усекается. В данном примере будет видна только середина длинного текста, так как текст выровнен по центру.

```
var s:string;
    X1,Y1,X2,Y2:integer;
```

```
s:=...
```

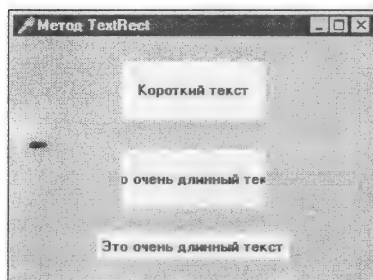
```
X1:=100;  
Y1:=100;  
X2:=200;  
Y2:=150;  
with Canvas do  
begin  
  Brush.Color:=clRed;  
  TextRect (Rect(X1,Y1,X2,Y2), X1+(X2-X1-TextWidth(s))  
div 2,  
            Y1+(Y2-Y1-TextHeight(s)) div 2,s);  
end;  
end;
```

Если в приведенном примере заменить оператор **TextRect** на `TextRect(Rect(X1-5,Y1-5, X1+TextWidth(st)+5,
Y1+TextHeight(st)+5), X1, Y1, st);`

то текст будет выводиться полностью в красной прямоугольной области, на 5 пикселей отступающей во все стороны от текста. Именно этим отступом, делающим надпись более красивой, этот оператор отличается от всех предыдущих более простых операторов. Примеры вывода текста методом **TextRect** показаны на рис. 3.3. Вверху относительно короткий текст выровнен по середине прямоугольной рамки, в которой он рисуется. Ниже в аналогичной по размерам рамке помещен более длинный текст: «Это очень длинный текст». Вы видите, что текст урезан по размерам рамки. Внизу вы видите вывод, сделанный оператором, аналогичным приведенному выше и обеспечивающим зазор в 5 пикселей во все стороны от надписи любой длины. Очевидно, что это наиболее удачный вариант.

Рис. 3.3

Примеры вывода текста методом **TextRect**



В целом можно сказать, что прямой вывод текста на канву описанными методами имеет смысл использовать только в исключительных случаях. Таким исключительным случаем является вывод каких-то надписей в компонент **Image** (см. раздел 4.3). Этот компонент предназначен для отображения графики. Но в тех случаях, когда на графику (на какую-то картинку) требуется

наложить текст, этот текст целесообразно выводить описанными методами. В остальных случаях это делать не имеет смысла, поскольку при прямом выводе текста на канву теряются преимущества визуального проектирования и приходится рассчитывать координаты размещения надписи.

3.3 Компоненты ввода и отображения текстовой информации

3.3.1 Перечень компонентов ввода и отображения текстовой информации

В библиотеке визуальных компонентов Delphi существует множество компонентов, позволяющих отображать, вводить и редактировать текстовую информацию. В таблице 3.1 приведен их перечень для Delphi с краткими характеристиками и указанием основных параметров, содержащих отображаемый или вводимый текст. В этой таблице не указаны аналогичные элементы отображения и редактирования текстов, содержащихся в базах данных, так как их рассмотрение выходит за рамки данной книги.

Таблица 3.1. Компоненты ввода и отображения текстовой информации

Пиктограмма	Компонент	Страница	Описание
	Label (метка)	Standard	Отображение текста, который не изменяется пользователем. Никакого оформления текста не предусмотрено, кроме цвета метки и текста. Основное свойство — Caption .
	StaticText (метка с бордюром)	Additional	Подобен компоненту Label , но обеспечивает возможность задания стиля бордюра. Основное свойство — Caption .
	Panel (панель)	Standard	Компонент является контейнером для группирования органов управления, но может использоваться и для отображения текста с возможностями объемного оформления. Основное свойство — Caption .

Пикто- грамма	Компонент	Страница	Описание
	Edit (окно редак- тирования)	Standard	Отображение, ввод и редактиро- вание однострочных текстов. Имеется возможность оформле- ния объемного бордюра. Основ- ное свойство — Text .
	MaskEdit (окно маски- рованного ре- дактирования)	Additional	Используется для форматирова- ния данных или для ввода сим- волов в соответствии с шабло- ном. Основные свойства — Text и EditText .
	Memo (многостроч- ное окно ре- дактирования)	Standard	Отображение, ввод и редактиро- вание многострочных текстов. Имеется возможность оформле- ния объемного бордюра. Основ- ное свойство — Lines .
	RichEdit (многостроч- ное окно ре- дактирования в формате RTF)	Win32	Компонент представляет собой окно редактирования в стиле Windows 95 в обогащенном фор- мате RTF, позволяющее произ- водить выбор атрибутов шриф- та, поиск текста и многое дру- гое. Основное свойство — Lines .
	ListBox (окно списка)	Standard	Отображение стандартного окна списка Windows, позволяющего пользователю выбирать из него пункты. Основное свойство — Items .
	CheckListBox (список с ин- дикаторами)	Additional	Компонент является комбина- цией свойств списка ListBox и индикаторов CheckBox в одном компоненте.
	ComboBox (редактируе- мый список)	Standard	Объединяет функции ListBox и Edit . Пользователь может либо ввести текст, либо выбрать его из списка. Основное свойство — Items .
	StringGrid (таблица строк)	Additional	Отображения текстовой инфор- мации в таблице из строк и столбцов с возможностью пере- мещаться по строкам и столб- цам и осуществлять выбор. Основное свойство — Cells .

В последующих разделах мы рассмотрим возможности и методику применения этих компонентов.

3.3.2 Отображение текста в надписях компонентов **Label**, **StaticText** и **Panel**

Для отображения различных надписей на форме используются в основном компоненты **Label**, **StaticText** (появившийся только в Delphi 3) и **Panel**. Первые два из этих компонентов — метки, специально предназначенные для отображения текстов. Основное назначение панели **Panel** другое: компоновка компонентов в окне формы. Однако, панель можно использовать и для вывода текстов.

Примеры вывода текста в компоненты приведены на рис. 3.4.

Тексты, отображаемые в перечисленных компонентах, определяются значением их свойства **Caption**. Его можно устанавливать в процессе проектирования или задавать и изменять программно во время выполнения приложения. Например:

```
Label1.Caption := 'Новый текст';
```

Если требуется отобразить числовую информацию, можно воспользоваться функциями **FloatToStr** и **IntToStr**, переводящими соответственно числа с плавающей запятой и целые в строку. Для формирования текста, состоящего из нескольких фрагментов, можно использовать операцию «+», которая для строк означает их склеивание (конкатенацию). Например, если в программе имеется целая переменная **I**, отображающая число сотрудников некоторой организации, то вывести в метку **Label1** информацию об этом можно оператором:

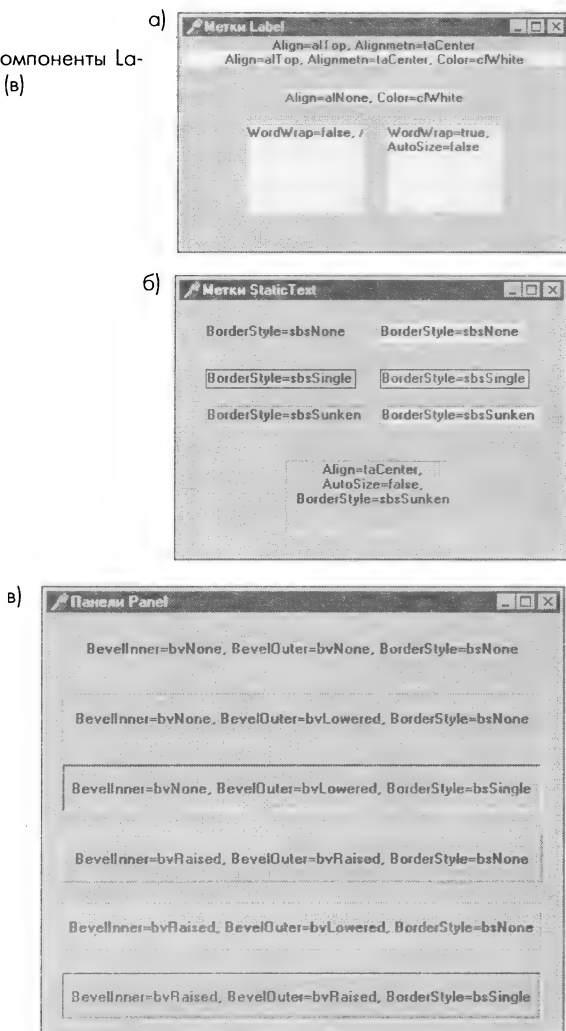
```
Label1.Caption := 'Число сотрудников: '+IntToStr(I);
```

Во всех компонентах цвет фона определяется свойством **Color**, а цвет надписи — подсвойством **Color** свойства **Font**. Например, в большинстве меток (кроме верхней) на рис. 3.4 а и в правых метках на рис. 3.4 б задан цвет фона равным **clWhite** — белый. Если цвет специально не задавать, то цвет фона обычно сливается с цветом контейнера, содержащего метку, так что фон просто не заметен.

Для метки **Label** цвет и шрифт — единственно доступные элементы оформления надписи. Компоненты **StaticText** и **Panel** имеют кроме того свойство **BorderStyle**, определяющее рамку текста — бордюр. На рис. 3.4 б вы можете видеть влияние бордюра на вид метки **StaticText**. При стиле **sbsNone** метка **StaticText** по виду не отличается от метки **Label**. Вероятно, если уж использовать бордюр, то наиболее приятный стиль **sbsSunken**.

Рис. 3.4

Примеры вывода текста в компоненты Label (а), StaticText (б) и Panel (в)



Компонент **Panel** кроме свойства **BorderStyle** имеет еще свойства **BevelInner**, **BevelOuter**, **BevelWidth**, **BorderWidth**, которые предоставляют богатые возможности оформления надписи, как вы можете видеть на рис. 3.4 в. Таким образом, с точки зрения оформления выводимого текста максимальные возможности дает **Panel** и минимальные — **Label**.

Размещение всех рассматриваемых компонентов на форме определяется, в частности, свойствами **Top** (координата верхнего

края), **Left** (координата левого края), **Height** (высота), **Width** (ширина). Имеются также свойства, определяющие изменение размера компонента при изменении пользователем во время выполнения приложения размеров окна. Это общие свойства всех оконных компонентов: **Aline** (выравнивание компонента по всей верхней, левой, правой, нижней частям контейнера или по всей его клиентской области), **Anchors** (привязка сторон компонента к сторонам контейнера), **Constraints** (ограничения допустимых изменений размеров). Учтите, что использование в свойстве **Anchors** одновременной привязки компонента к противоположным сторонам контейнера приведет к растягиванию или сжатию компонента при изменении пользователем размеров окна. При этом в метках в ряде случаев наблюдаются неприятности, связанные с перемещением надписей (если они, например, выровнены по центру). Чтобы избежать этих неприятностей, надо в обработку события формы **OnResize** вставить операторы, перерисовывающие компоненты методом **Repaint**, например:

```
StaticText1.Repaint;
```

Размер меток **Label** и **StaticText** определяется также свойством **AutoSize**. Если это свойство установлено в **true**, то вертикальный и горизонтальный размеры компонента определяются размером надписи. Если же **AutoSize** равно **false**, то выравнивание текста внутри компонента определяется свойством **Alignment**, которое позволяет выравнивать текст по левому краю, правому краю или центру клиентской области метки. В панели **Panel** также имеется свойство **AutoSize**, но оно не относится к размерам надписи **Caption**. Однако, свойство выравнивания **Alignment** работает и для панели.

В метке **Label** имеется свойство **WordWrap** — допустимость переноса слов длинной надписи, превышающей длину компонента, на новую строчку. Чтобы такой перенос мог осуществляться, надо установить свойство **WordWrap** в **true**, свойство **AutoSize** в **false** (чтобы размер компонента не определялся размером надписи) и сделать высоту компонента такой, чтобы в нем могло поместиться несколько строк (см. пример правой нижней метки на рис. 3.4 а). Если **WordWrap** не установлено в **true** при **AutoSize** равном **false**, то длинный текст, не помещающийся в рамке метки, просто обрезается (см. пример левой нижней метки на рис. 3.4 а).

В метке **StaticText** перенос длинного текста осуществляется автоматически, если значение **AutoSize** установлено в **false** и размер компонента достаточен для размещения нескольких строк. Для того, чтобы в **StaticText** осуществлялся перенос при изменении пользователем размеров окна, надо осуществлять описанную

выше перерисовку компонента методом **Repaint** в обработчике события формы **OnResize**.

В панели размещение надписи в нескольких строках невозможно.

Можно отметить еще одно свойство меток **Label** и **StaticText**, превращающее их в некоторое подобие управляющих элементов. Это свойство **FocusControl** — фокусируемый компонент. Если в свойстве метки **Caption** поместить перед одним из символов символ амперсанта «&», то символ, перед которым поставлен амперсанта, отображается в надписи метки подчеркнутым (сам амперсанта вообще не отображается). Если после этого обратиться к свойству метки **FocusControl**, то из выпадающего списка можно выбрать элемент, на который будет переключаться фокус, если пользователь нажмет клавиши ускоренного доступа: клавишу **Alt** + подчеркнутый символ. Подобные клавиши ускоренного доступа предусмотрены в управляющих элементах: разделах меню (см. раздел 6.1) и кнопках (см. раздел 5.2). Благодаря свойству **FocusControl** метки могут обеспечить клавишами ускоренного доступа иные элементы, например, окна редактирования (см. раздел 3.3.3), в которых такие клавиши не предусмотрены. Только для того, чтобы клавиши ускоренного доступа в метках срабатывали, необходимо установить свойство **ShowAccelChar** этих меток в **true**.

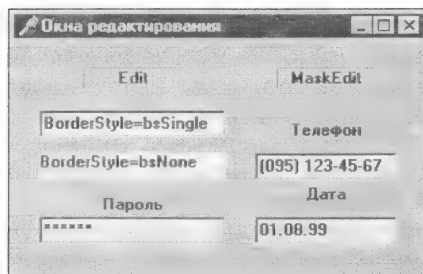
Для отображение текстовой информации, и даже с дополнительной возможностью прокрутки длинных текстов, можно использовать также окна редактирования **Edit** и **MaskEdit** в режиме **ReadOnly**.

3.3.3 Окна редактирования **Edit** и **MaskEdit**

На рис. 3.5 вы можете увидеть примеры окон редактирования. Внешнее оформление окон редактирования определяется свойством **BorderStyle**, влияние которого на вид компонента вы можете увидеть на том же рисунке.

Рис. 3.5

Примеры окон редактирования



В компонентах **Edit** и **MaskEdit** вводимый и выводимый текст содержится в свойстве **Text**. Это свойство можно устанавливать в процессе проектирования или задавать программно. Выравнивание текста, как это имело место в метках и панелях, невозможно. Перенос строк тоже невозможен. Текст, не помещающийся по длине в окно, просто сдвигается и пользователь может перемещаться по нему с помощью курсора. Свойство **AutoSize** в окнах редактирования имеет смысл, отличный от смысла аналогичного свойства меток: автоматически подстраивается под размер текста только высота, но не ширина окна.

Окна редактирования снабжены многими функциями, свойственными большинству редакторов. Например, в них предусмотрены типичные комбинации «горячих» клавиш: **Ctrl-C** — копирование выделенного текста в буфер обмена **Clipboard** (команда **Copy**), **Ctrl-X** — вырезание выделенного текста в буфер **Clipboard** (команда **Cut**), **Ctrl-V** — вставка текста из буфера **Clipboard** в позицию курсора (команда **Paste**), **Ctrl-Z** — отмена последней команды редактирования. Правда, пользователи часто не догадываются об этих возможностях редактирования. Так что полезно напоминать им об этом соответствующими подсказками.

Свойство **AutoSelect** определяет, будет ли автоматически выделяться весь текст при передаче фокуса в окно редактирования. Его имеет смысл задавать равным **true** в случаях, когда при переключении в данное окно пользователь будет скорее заменять текущий текст, чем исправлять его. Имеются также свойства только времени выполнения **SelLength**, **SelStart**, **SelText**, определяющие соответственно длину выделенного текста, позицию перед первым символом выделенного текста и сам выделенный текст. Например, если в окне имеется текст «выделение текста» и в нем пользователь выделил слово «текста», то **SelLength** = 6, **SelStart** = 10 и **SelText** = «текста». Если выделенного текста нет, то свойство **SelStart** просто определяет текущее положение курсора.

Окна редактирования можно использовать и просто как компоненты отображения текста. Для этого надо установить в **false** их свойство **ReadOnly** и целесообразно установить **AutoSelect** в **false**. В этом случае пользователь не сможет изменять отображаемый текст и окно редактирования становится подобным меткам, рассмотренным в разделе 3.3.2. Но имеются и определенные отличия. Во-первых, окна редактирования оформлены несколько иначе (сравните рис. 3.4 и 3.5.). А главное — окна редактирования могут вмещать текст, превышающий их длину. В этом случае пользователь может прокручивать этот текст, перемещая курсор в окне. Такими особенностями не обладает ни одна метка.

При использовании окон редактирования для вывода, ввода и редактирования чисел необходимо использовать функции взаимного преобразования строк и чисел. Для вывода это описанные при рассмотрении меток функции **FloatToStr** и **IntToStr**. При вводе это функции **StrToFloat** — преобразование строки в значение с плавающей запятой, и **StrToInt** — преобразование строки в целое значение. Если вводимый текст не соответствует числу (например, содержит недопустимые символы), то функции преобразования генерируют исключение **EConvertError**. Поэтому в программе необходимо предусмотреть обработку этого исключения. Например:

```
var A: integer;
.....
try
    .....
    A := StrToInt(Edit1.Text);
    ... {операторы, использующие переменную A}
except
    on EConvertError do
        ShowMessage('Вы ввели ошибочное число;
                    повторите ввод');
end;
```

Этот код обеспечивает сообщение пользователю об ошибке ввода и предотвращает ошибочные вычисления. Впрочем, это не лучший вариант предотвратить ошибочный ввод, поскольку пользователь узнает о своей ошибке только после того, как программа пытается использовать введенные данные. Лучше, если пользователь просто не сможет ввести неправильные символы. Например, если вы хотите, чтобы пользователь мог вводить в окно редактирования **Edit** только цифры и символ точки, вы можете в обработчик события **OnKeyPress** этого компонента вставить оператор:

```
if not (Key in ['0'..'9', '.', ',']) then Key := #0;
```

Этот оператор подменит все символы, кроме цифр и запятой, нулевым символом, который не занесется в текст окна **Edit**.

Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength** = 0, то длина текста не ограничена. В противном случае значение **MaxLength** указывает максимальное число символов, которое может ввести пользователь.

Свойство **Modified**, доступное только во время выполнения, показывает, проводилось ли редактирование текста в окне. Если вы хотите использовать это свойство, то в момент начала работы пользователя с текстом **Modified** надо установить в **false**. Тогда при последующем обращении к этому свойству можно по его зна-

чению (**true** или **false**) установить, было или не было произведено редактирование.

Свойство **PasswordChar** позволяет превращать окно редактирования в окно ввода пароля. По умолчанию значение **PasswordChar** равно **#0** — нулевому символу. В этом случае это обычное окно редактирования. Но если в свойстве указать иной символ (например, символ звездочки «*»), то при вводе пользователем текста в окне будут появляться именно эти символы, а не те, которые вводит пользователь (см. рис. 3.5). Тем самым обеспечивается секретность ввода пароля.

Компонент **MaskEdit** отличается от **Edit** тем, что в нем можно задать строку маски в свойстве **EditMask**. Маска состоит из трех разделов, между которыми ставится точка с запятой (;). В первом разделе — шаблоне записываются специальным образом символы (см. таблицу 3.2), которые можно вводить в каждой позиции, и символы, добавляемые самой маской; во втором разделе записывается 1 или 0 в зависимости от того, надо или нет, чтобы символы, добавляемые маской, включались в свойство **Text** компонента; в третьем разделе указывается символ, используемый для обозначения позиций, в которых еще не осуществлен ввод. Прочитать результат ввода можно или в свойстве **Text**, которое в зависимости от вида второго раздела маски включает или не включает в себя символы маски, или в свойстве **EditText**, содержащем введенный текст вместе с символами маски.

Таблица 3.2. Символы шаблона маски

!	Наличие символа «!» означает, что в EditText недостающие символы <u>предваряются</u> пробелами, а отсутствие символа «!» означает, что пробелы размещаются <u>в конце</u> .
>	Символ «>» означает, что все последующие за ним символы должны вводиться в верхнем регистре, пока не кончится маска или пока не встретится символ «<».
<	Символ «<» означает, что все последующие за ним символы должны вводиться в нижнем регистре, пока не кончится маска или пока не встретится символ «>».
<>	Символы «<>» означают, что анализ регистра не производится.
\	Символ «\» означает, что следующий за ним символ является буквенным, а не специальным, характерным для маски. Например, символ «>» после символа «\» воспримется как знак >, а не как символ, указывающий на верхний регистр.

L	Символ «L» означает, что в данной позиции должна быть буква.
l	Символ «l» означает, что в данной позиции может быть только буква или ничего.
A	Символ «A» означает, что в данной позиции должна быть буква или цифра.
a	Символ «a» означает, что в данной позиции может быть буква, или цифра, или ничего.
C	Символ «C» означает, что в данной позиции должен быть любой символ.
c	Символ «c» означает, что в данной позиции может быть любой символ или ничего.
0	Символ «0» означает, что в данной позиции должна быть цифра.
9	Символ «9» означает, что в данной позиции может быть цифра или ничего.
#	Символ «#» означает, что в данной позиции может быть цифра, знак «+», знак «-» или ничего.
:	Символ «:» используется для разделения часов, минут и секунд.
/	Символ «/» используется для разделения месяцев, дней и годов в датах.
	Символ « » означает автоматическую вставку в текст пробела.

Вводить маску можно непосредственно в свойство **EditMask**. Но удобнее пользоваться специальным редактором масок, вызываемым при нажатии кнопки с многоточием в строке свойства **EditMask** в Инспекторе Объектов. Окно редактора масок имеет вид, представленный на рис. 3.6.

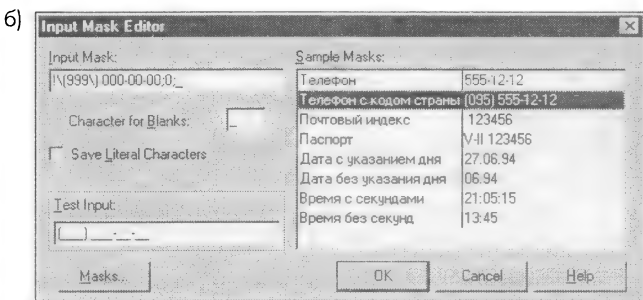
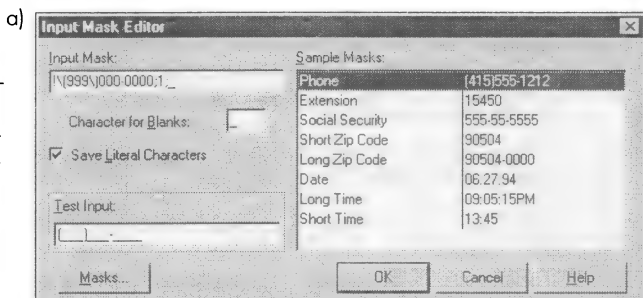
В редакторе масок окно **Sample Masks** содержит наименования стандартных масок и примеры ввода с их помощью. В окно **Input Mask** надо ввести маску. Если вы выбираете одну из стандартных масок, то окно **Input Mask** автоматически заполняется и вы можете, если хотите, отредактировать эту маску.

Окно **Character for Blanks** определяет символ, используемый для обозначения позиций, в которых еще не осуществлен ввод (третий раздел маски). Индикатор **Save Literal Characters** определяет второй раздел маски: установлен, если второй раздел равен 1; и не установлен, если второй раздел равен 0.

Кнопка **Masks** позволяет выбрать и загрузить какой-либо другой файл стандартных масок. К сожалению, среди файлов стан-

Рис. 3.6

Окно редактора масок с загруженными файлами стандартных масок: американским (а) и российским (б)



дартных масок, поставляемых с Delphi, отсутствует маска, соответствующая российским стандартам. Но вы легко можете сами сделать себе такой файл стандартных масок. Он делается в обычном текстовом редакторе и должен сохраняться как «только текст» с расширением **.dem**. Чтобы редактор масок Delphi видел этот файл, его надо сохранить в каталоге Delphi **BIN**. Каждая строка файла состоит из трех частей, разделяемых символом вертикальной черты. Первая часть состоит из пояснительного текста, появляющегося в левой панели окна Sample Masks редактора масок. Вторая часть — пример, который появляется в правой панели окна Sample Masks редактора масок. А третья часть — сама маска. Например, я сделал себе файл с текстом, приведенным ниже, и сохранил его с именем **ru.dem**.

```
Телефон | 5551212 | !000-00-00;0;_
Телефон с кодом страны | 0955551212 | !\ (999\
000-00-00;0;_
Почтовый индекс | 123456 | !0000000;1;_
Паспорт | VII123456 | !L-LL 999999;0;_
Дата с указанием дня | 270694 | !99/99/00;1;_
Дата без указания дня | 0694 | !99/00;1;_
Время с секундами | 210515 | !90:00:00;1;_
Время без секунд | 1345 | !90:00;1;_
```

На рис. 3.6 б вы можете видеть его в работе, а на рис. 3.5 вы можете видеть ввод в окна с масками телефона и даты.

Рассмотрим примеры масок. В приведенном выше файле маска для ввода номера телефона имеет вид:

```
!\(999\) 000-00-00;0;_
```

В этой маске символ **9** означает, что в соответствующей позиции может быть только цифра. Символ **0** означает, что в данной позиции должна быть цифра. Символ подчеркивания в конце маски будет заполнять пустые позиции. Таким образом, пользователю для ввода в окне будет отображен шаблон (см. рис. 3.5):

```
( ) _ - _ - _
```

Поскольку второй раздел маски равен **0**, то при чтении введенных пользователем значений свойства **EditText** и **Text** будут различаться. Свойство **EditText** для примера рис. 3.5 будет равно **(095) 123-45-67**, а свойство **Text** будет равно **0951234567**. Если второй раздел маски сделать равным **1**, то значения обоих свойств будут равны **(095) 123-45-67**.

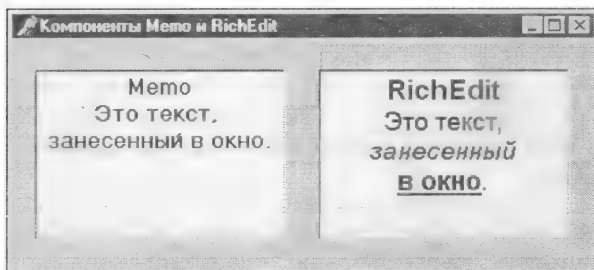
Рассмотрим еще пример. Если с помощью **EditMask** надо ввести, например, целое число без знака, состоящее не более, чем из двух цифр, можно задать маску **99;0;**. Если число обязательно должно быть двузначным, то маска должна иметь вид **00;0;**.

3.3.4 Многострочные окна редактирования Мемо и RichEdit

Компоненты **Memo** и **RichEdit** (см. пример на рис. 3.7) являются окнами редактирования многострочного текста. Они так же, как и окно **Edit**, снабжены многими функциями, свойственными большинству редакторов. В них предусмотрены типичные комбинации «горячих» клавиш: **Ctrl-C** — копирование выделенного текста в буфер обмена **Clipboard** (команда **Copy**), **Ctrl-X** — вырезание выделенного текста в буфер **Clipboard** (команда **Cut**), **Ctrl-V** — вставка текста из буфера **Clipboard** в позицию курсора (команда **Paste**), **Ctrl-Z** — отмена последней команды редактирования.

Рис. 3.7

Примеры компонентов **Memo** и **RichEdit**



В компоненте **Memo** формат (шрифт, его атрибуты, выравнивание) одинаков для всего текста и определяется свойством **Font**. Если вы сохраните в файле текст, введенный или отредактированный пользователем, то будет создан текстовый файл, содержащий только символы и не содержащий элементов форматирования. При последующем чтении этого файла в **Memo** формат будет определяться текущим состоянием свойства **Font** компонента **Memo**, а не тем, в каком формате ранее вводился текст.

Компонент **RichEdit** работает с текстом в обогащенном формате RTF. При желании изменить атрибуты вновь вводимого фрагмента текста вы можете задать свойство **SelAttributes**. Это свойство типа **TTextAttributes**, которое в свою очередь имеет подсвойства: **Color** (цвет), **Name** (имя шрифта), **Size** (размер), **Style** (стиль) и ряд других. Например, введите на форму компонент **RichEdit**, диалог выбора шрифта **FontDialog** (см. раздел 8.4) и кнопку **Button**, которая позволит пользователю менять атрибуты текста. В обработчик щелчка кнопки можно было бы ввести текст:

```
if FontDialog1.Execute then
  with RichEdit1.SelAttributes do begin
    Color:=FontDialog1.Font.Color;
    Name:=FontDialog1.Font.Name;
    Size:=FontDialog1.Font.Size;
    Style:=FontDialog1.Font.Style;
  end;
RichEdit1.SetFocus;
```

В приведенном коде присваивается поочередно значение каждого свойства. Но этот текст можно кардинально сократить, воспользовавшись тем, что объекты **SelAttributes** и **Font** совместимы по типу. Поэтому можно присвоить сразу все свойства одного объекта другому:

```
if FontDialog1.Execute then
  RichEdit1.SelAttributes.Assign(FontDialog1.Font);
RichEdit1.SetFocus;
```

Запустите приложение и увидите, что вы можете менять атрибуты текста, выполняя отдельные фрагменты различными шрифтами, размерами, цветами, стилями. Устанавливаемые атрибуты влияют на выделенный текст или, если ничего не выделено, то на атрибуты нового текста, вводимого начиная с текущей позиции курсора (позиция курсора определяется свойством **SelStart**).

В компоненте имеется также свойство **DefAttributes**, содержащее атрибуты по умолчанию. Эти атрибуты действуют до того момента, когда изменяются атрибуты в свойстве **SelAttributes**.

Но значения атрибутов в **DefAttributes** сохраняются и в любой момент эти значения могут быть методом **Assign** присвоены атрибутам свойства **SelAttributes**, чтобы вернуться к прежнему стилю.

Свойство **DefAttributes** доступно только во время выполнения. Поэтому его атрибуты при необходимости можно задавать, например, в обработчике события **OnCreate**.

За выравнивание, отступы и т.д. в пределах текущего абзаца отвечает свойство **Paragraph** типа **TParaAttributes**. Этот тип в свою очередь имеет ряд свойств:

Alignment	Определяет выравнивание текста. Может принимать значения taLeftJustify (влево), taCenter (по центру) или taRightJustify (вправо).
FirstIndent	Число пикселей отступа красной строки.
Numbering	Управляет вставкой маркеров, как в списках. Может принимать значения nsNone — отсутствие маркеров, nsBullet — маркеры ставятся.
LeftIndent	Отступ в пикселях от левого поля.
RightIndent	Отступ в пикселях от правого поля.
TabCount	Количество позиций табуляции.
Tab	Значения позиций табуляции в пикселях.

Значения подсвойств свойства **Paragraph** можно задавать только в процессе выполнения приложения, например, в событии создания формы или при нажатии какой-нибудь кнопки. Значения подсвойств свойства **Paragraph** относятся к тому абзацу, в котором находится курсор. Например, каждый из следующих операторов осуществит соответственное выравнивание текущего абзаца:

```
RichEdit1.Paragraph.Alignment:=taLeftJustify; // Влево
RichEdit1.Paragraph.Alignment:=taCenter;      // По центру
RichEdit1.Paragraph.Alignment:=taRightJustify; // Вправо
```

Следующий оператор приведет к тому, что текущий абзац будет отображаться как список, т.е. с маркерами:

```
RichEdit1.Paragraph.Numbering:=nsBullet;
```

Уничтожение списка в текущем абзаце осуществляется оператором

```
RichEdit1.Paragraph.Numbering:=nsNone;
```

В целом, если с помощью компонента **ActionList** (см. раздел 9.2) определено некоторое действие ввода и уничтожения списка, названное **ABullet**, то операторы обработки соответствующего действия могут иметь вид:

```
if (ABullet.Checked) then  
    RichEdit1.Paragraph.Numbering:=nsNone  
    else RichEdit1.Paragraph.Numbering:=nsBullet;  
ABullet.Checked:=not ABullet.Checked;
```

Они обеспечивают переключение соответствующей быстрой кнопки и раздела меню из нажатого состояния (отмеченного) в ненажатое с соответствующим изменением вида текущего абзаца.

Свойства **TabCount** и **Tab** имеют смысл при вводе текста только при значении свойства компонента **WantTabs = true**. Это свойство разрешает пользователю вводить в текст символ табуляции. Если **WantTabs = false**, то нажатие пользователем клавиши табуляции просто переключит фокус на очередной компонент и символ табуляции в текст не введется.

Мы рассмотрели основные отличия **Memo** и **RichEdit**. Теперь остановимся на общих свойствах этих окон редактирования.

Свойства **Alignment** и **WordWrap** имеют тот же смысл, что, например, в метках, и определяют выравнивание текста и допустимость переноса длинных строк. Установка свойства **ReadOnly** в **true** задает текст только для чтения. Свойство **MaxLength** определяет максимальную длину вводимого текста. Если **MaxLength = 0**, то длина текста не ограничена. Свойства **WantReturns** и **WantTab** определяют допустимость ввода пользователем в текст символов перевода строки и табуляции.

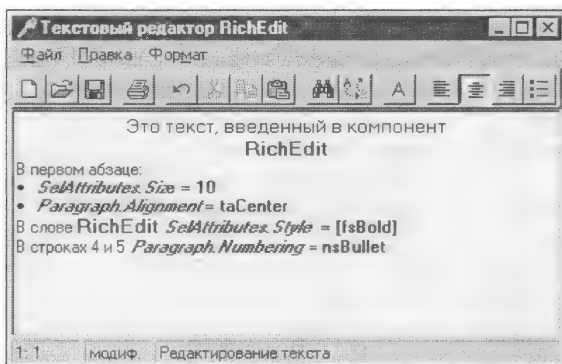
Свойство **ScrollBars** определяет наличие полос прокрутки текста в окне. По умолчанию **ScrollBars = ssNone**, что означает их отсутствие. Пользователь может в этом случае перемещаться по тексту только с помощью курсора. Можно задать свойству **ScrollBars** значения **ssHorizontal**, **ssVertical** или **ssBoth**, что будет соответственно означать наличие горизонтальной, вертикальной или обеих полос прокрутки.

В качестве примера на рис. 3.8 приведен пример текстового редактора, использующего описанные выше свойства компонента **RichEdit**. Текст в окне редактора частично поясняет атрибуты шрифта, использованные при его написании.

Основное свойство окон **Memo** и **RichEdit** — **Lines**, содержащее текст окна в виде списка строк и имеющее тип **TStrings**. Начальное значение текста можно установить в процессе проектирования, нажав кнопку с многоточием около свойства **Lines** в окне Инспектора Объектов. Перед вами откроется окно редакти-

Рис. 3.8

Пример редактора на основе компонента RichEdit

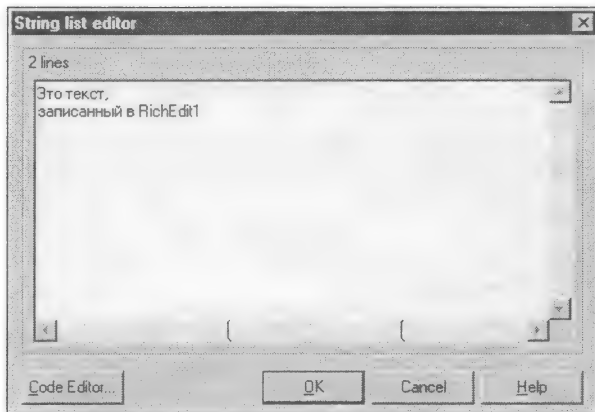


рования списков строк, представленное на рис. 3.9. Вы можете редактировать или вводить текст непосредственно в этом окне, или нажать кнопку CodeEditor и работать в обычном окне Редактора Кода. В этом случае, завершив работу с текстом, выберите из контекстного меню, всплывающего при щелчке правой кнопкой мыши, команду Close Page и ответьте утвердительно на вопрос, хотите ли вы сохранить текст в соответствующем свойстве окна редактирования.

В Delphi 1 аналогичное окно имеет еще кнопки, позволяющие загрузить текст из файла или сохранить его в файле.

Рис. 3.9

Окно редактирования списков строк



Во время выполнения приложения вы можете заносить текст в окно редактирования с помощью методов свойства **Lines** типа **TStrings**. Этот тип широко используется в свойствах многих компонентов и его описание вы можете найти в во встроенной

справке Delphi. Здесь коротко укажем только на его основные свойства и методы, используемые в свойстве **Lines**.

Весь текст, представленный одной строкой типа **String**, внутри которой используются разделители типа символов возврата каретки и перевода строки, содержится в свойстве **Text**.

Доступ к отдельной строке текста вы можете получить с помощью свойства **Strings[Index: Integer]**. Индексы, как и везде в Delphi, начинаются с 0. Так что **Memo1.Lines.Strings[0]** — это текст первой строки. Учтите, что если окно редактирования изменяется в размерах при работе с приложением и свойство **WordWrap = true**, то индексы строк будут изменяться при переносах строк, так что в этих случаях индекс мало о чем говорит.

Свойство только для чтения **Count** указывает число строк в тексте.

Для очистки текста в окне надо выполнить процедуру **Clear**. Этот метод относится к самому окну, а не к его свойству **Lines**.

Для занесения новой строки в конец текста окна редактирования можно воспользоваться методами **Add** или **Append** свойства **Lines**. Для загрузки текста из файла применяется метод **LoadFromFile**. Сохранение текста в файле осуществляется методом **SaveToFile**.

Пусть, например, в вашем приложении имеется окно редактирования **Edit1**, в котором пользователь вводит имя сотрудника, и есть кнопка, при щелчке на которой в окно **Memo1** должна занестись шапка характеристики этого сотрудника, после чего пользователь может заполнить текст характеристики.

Обработчик щелчка на кнопке может иметь вид:

```
Memo1.Clear;  
Memo1.Lines.Add('Х А Р А К Т Е Р И С Т И К А');  
Memo1.Lines.Add('Сотрудник '+Edit1.Text);  
Memo1.SetFocus;
```

Загрузка в окно **Memo1** текста из файла (например, хранящейся в файле характеристики сотрудника) может осуществляться командой

```
Memo1.Lines.LoadFromFile('text.txt');
```

Сохранение текста в файле может осуществляться командой

```
Memo1.Lines.SaveToFile('text.txt');
```

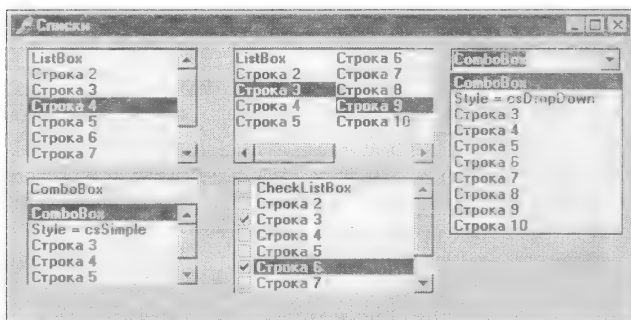
Свойство **SelStart** компонентов **Memo** и **Richedit** указывает позицию курсора в тексте или начало выделенного пользователем текста. Свойство **CaretPos** указывает на запись, поле **X** которой содержит индекс символа в строке, перед которым расположен курсор, а поле **Y** — индекс строки, в которой находится курсор (встроенная справка Delphi утверждает другое — что свойст-

во **CaretPos** содержит координаты курсора в пикселях; но, к счастью, это не так). Таким образом, учитывая, что индексы начинаются с 0, значения **Memo1.CaretPos.Y+1** и **Memo1.CaretPos.X+1** определяют соответственно номер строки и символа в ней, перед которым расположен курсор. В редакторе на рис. 3.8 именно эти значения (только не для **Memo**, а для **RichEdit**) использованы, чтобы отображать в строке состояния (см. раздел 7.6) позицию курсора.

3.3.5 Компоненты выбора из списков — **ListBox**, **CheckBox** и **ComboBox**

Пример компонентов, рассмотренных в данном разделе и обеспечивающих выбор из списка, приведен на рис. 3.10.

Рис. 3.10
Пример
компонентов
выбора
из списков



Компоненты **ListBox** и **ComboBox** отображают списки строк. Они отличаются друг от друга прежде всего тем, что **ListBox** только отображает данные и позволяет пользователю выбрать из них то, что ему надо, а **ComboBox** позволяет также редактировать данные. Кроме того различается форма отображения списков. **ListBox** отображает список в раскрытом виде и автоматически добавляет в список полосы прокрутки, если все строки не помещаются в окне компонента. **ComboBox** позволяет отображать список как в развернутом виде, так и в виде выпадающего списка, что обычно удобнее, так как экономит площадь окна приложения.

Основное свойство обоих компонентов, содержащее список строк, — **Items**, имеющее рассмотренный ранее тип **TStrings**. Заполнить его во время проектирования можно, нажав кнопку с многоточием около этого свойства в окне Инспектора Объектов. Во время выполнения работать с этим свойством можно, пользуясь свойствами и методами класса **TStrings** (см. раздел 3.3.4) — **Clear**, **Add** и другими.

В компоненте **ListBox** имеется свойство **MultiSelect**, разрешающее пользователю множественный выбор в списке (на рис. 3.10 это свойство установлено в **true** в среднем верхнем списке). Если **MultiSelect = false** (значение по умолчанию), то пользователь может выбрать только один элемент списка. В этом случае можно узнать индекс выбранной строки из свойства **ItemIndex**, доступного только во время выполнения. Если ни одна строка не выбрана, то **ItemIndex = -1**. Начальное значение **ItemIndex** невозможно задать во время проектирования. По умолчанию **ItemIndex = -1**. Это означает, что ни один элемент списка не выбран. Если вы хотите задать этому свойству какое-то другое значение, т.е. установить выбор по умолчанию, который будет показан в момент начала работы приложения, то сделать это можно, например, в обработчике события **OnCreate** формы, введя в него оператор вида

```
ListBox1.ItemIndex:=0;
```

Если допускается множественный выбор (**MultiSelect = true**), то значение **ItemIndex** соответствует тому элементу списка, который находится в фокусе. При множественном выборе проверить, выбран ли данный элемент, можно проверив свойство **Selected[Index: Integer]** типа **Boolean**.

На способ множественного выбора при **MultiSelect = true** влияет еще свойство **ExtendedSelect**. Если **ExtendedSelect = true**, то пользователь может выделить интервал элементов, выделив один из них, затем нажав клавишу **Shift** и переведя курсор к другому элементу. Выделить не прилегающие друг к другу элементы пользователь может, если будет удерживать во время выбора нажатой клавишу **Ctrl**. Если же **ExtendedSelect = false**, то клавиши **Shift** и **Ctrl** при выборе не работают.

Свойство **Columns** определяет число столбцов, в которых будет отображаться список, если он не помещается целиком в окне компонента **ListBox** (в среднем верхнем списке на рис. 3.10 свойство **Columns** равно 2).

Свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted = true** новые строки в список добавляются не в конец, а по алфавиту.

Свойство **Style**, установленное в **lbStandard** (значение по умолчанию) соответствует списку строк. Другие значения **Style** позволяют отображать в списке не только текст, но и изображения.

Имеется еще один компонент, очень похожий на **ListBox** — это список с индикаторами **CheckListBox**. Выглядит он так же, как **ListBox** (средний нижний список на рис. 3.10), но около

каждой строки имеется индикатор, который пользователь может переключать. Индикаторы можно переключать и программно, если список используется для вывода данных и необходимо в нем отметить какую-то характеристику каждого объекта, например, наличие товара данного наименования на складе.

Все свойства, характеризующие компонент **CheckListBox** как список, аналогичны **ListBox**, за исключением свойств, определяющих множественный выбор. Эти свойства компоненту **CheckListBox** не нужны, поскольку в нем множественный выбор можно осуществлять установкой индикаторов. Свойства компонента **CheckListBox**, связанные с индикаторами, будут рассмотрены в разделе 5.5.

Рассмотрим теперь компонент **ComboBox**. Стилль изображения этого компонента определяется свойством **Style**, которое может принимать следующие основные значения:

csDropDown	Выпадающий список со строками одинаковой высоты и с окном редактирования, позволяющим пользователю вводить или редактировать текст (правый список на рис. 3.10).
csSimple	Развернутый список со строками одинаковой высоты и с окном редактирования, позволяющим пользователю вводить или редактировать текст (левый нижний список на рис. 3.10).
csDropDownList	Выпадающий список со строками одинаковой высоты, не содержащий окна редактирования.

Выбор пользователя или введенный им текст можно определить по значению свойства **Text**. Если же надо определить индекс выбранного пользователем элемента списка, то можно воспользоваться обсуждавшимся в компоненте **ListBox** свойством **ItemIndex**. Все сказанное выше об **ItemIndex** и о задании его значения по умолчанию справедливо и для компонента **ComboBox**. Причем для **ComboBox** задание начального значения **ItemIndex** еще актуальнее, чем для **ListBox**. Если начальное значение не задано, то в момент запуска приложения пользователь не увидит в окне компонента одно из возможных значений списка и, вероятнее всего, не очень поймет, что с этим окном надо делать.

Если в окне проводилось редактирование данных, то **ItemIndex = -1**. По этому признаку можно определить, что редактирование проводилось.

Свойство **MaxLength** определяет максимальное число символов, которые пользователь может ввести в окно редактирования.

Если **MaxLength** = 0, то число вводимых символов не ограничено.

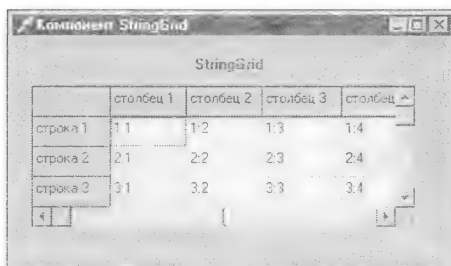
Как и в компоненте **ListBox**, свойство **Sorted** позволяет упорядочить список по алфавиту. При **Sorted** = **true** новые строки в список добавляются не в конец, а по алфавиту.

3.3.6 Таблица строк — компонент **StringGrid**

Компонент **StringGrid** (см. пример на рис. 3.11) представляет собой таблицу, содержащую строки. Данные таблицы могут быть только для чтения или редактируемы. Таблица может иметь полосы прокрутки, причем заданное число первых строк и столбцов может быть фиксированным и не прокручиваться. Таким образом, можно задать заголовки столбцов и строк, постоянно присутствующие в окне компонента. Каждой ячейке таблицы может быть поставлен в соответствие некоторый объект.

Рис. 3.11

Пример компонента **StringGrid**



Компонент **StringGrid** предназначен в первую очередь для отображения таблиц текстовой информации. Однако в разделе 4.4 поясняется, как этот компонент может отображать и графическую информацию.

Основные свойства компонента, определяющие отображаемый текст:

Cells[ACol, ARow: Integer]:
string

Строка, содержащаяся в ячейке с индексами столбца и строки **ACol** и **ARow**.

Cols[Index: Integer]: **TStrings**

Список строк, содержащихся в столбце с индексом **Index**.

Rows[Index: Integer]: **TStrings**

Список строк, содержащихся в строке с индексом **Index**.

Objects [ACol, ARow: Integer]:
TObject;

Объект, связанный со строкой, содержащейся в ячейке с индексами столбца и строки **ACol** и **ARow**.

Все эти свойства доступны во время выполнения. Задавать тексты можно программно или по отдельным ячейкам, или сразу по столбцам и строкам с помощью методов класса **TStrings**, описанных ранее в разделе 3.3.4.

Свойства **ColCount** и **RowCount** определяют соответственно число столбцов и строк, свойства **FixedCols** и **FixedRows** — число фиксированных, не прокручиваемых столбцов и строк. Цвет фона фиксированных ячеек определяется свойством **FixedColor**. Свойства **LeftCol** и **TopRow** определяют соответственно индексы первого видимого на экране в данный момент прокручиваемого столбца и первой видимой прокручиваемой строки.

Свойство **ScrollBars** определяет наличие в таблице полос прокрутки. Причем полосы прокрутки появляются и исчезают автоматически в зависимости от того, помещается таблица в соответствующий размер, или нет.

Свойство **Options** является множеством, определяющим многие свойства таблицы: наличие разделительных вертикальных и горизонтальных линий в фиксированных (**goFixedVertLine** и **goFixedHorzLine**) и не фиксированных (**goVertLine** и **goHorzLine**) ячейках, возможность для пользователя изменять с помощью мыши размеры столбцов и строк (**goColSizing** и **goRowSizing**), перемещать столбцы и строки (**goColMoving** и **goRowMoving**) и многое другое. Важным элементом в свойстве **Options** является **goEditing** — возможность редактировать содержимое таблицы.

В основном компонент **StringGrid** используется для выбора пользователем каких-то значений, отображенных в ячейках. Свойства **Col** и **Row** показывают индексы столбца и колонки выделенной ячейки. Возможно также выделение пользователем множества ячеек, строк и столбцов.

Среди множества событий компонента **StringGrid** следует отметить событие **OnSelectCell**, возникающее в момент выбора пользователем ячейки. В обработчик этого события передаются целые параметры **ACol** и **ARow** — столбец и строка выделенной ячейки, и булев параметр **CanSelect** — допустимость выбора. Параметр **CanSelect** можно использовать для запрета выделения ячейки, задав его значение **false**. А параметры **ACol** и **ARow** могут использоваться для какой-то реакции программы на выделение пользователя. Например, оператор

```
Label1.Caption:='Выбрана ячейка '
                    +IntToStr (ARow)+' ':''+IntToStr (ACol);
```

выдаст в метку **Label1** номер выбранной ячейки. А оператор

```
Label1.Caption:=StringGrid1.Cells[ACol, ARow];
```

выведет в ту же метку текст выделенной ячейки. Конечно, в реальном приложении задача заключается не в том, чтобы вывести подобные тексты при выборе пользователем той или иной ячейки, а в том, чтобы сделать нечто более полезное.

3.4 Ввод и отображение чисел, дат и времени

3.4.1 Перечень компонентов ввода и отображения чисел, дат и времени

В библиотеке визуальных компонентов Delphi существует ряд компонентов, позволяющих вводить, отображать и редактировать числа, даты и время. Конечно, с подобной информацией можно обращаться и просто как с текстовой, используя компоненты, описанные в разделе 3.3. Но это не удобно, так как не гарантирует от ошибок при вводе. В таблице 3.2 приведен перечень специализированных компонентов ввода и отображения чисел, дат и времени с краткими характеристиками и указанием основных параметров, содержащих отображаемый или вводимый текст. В этой таблице не указаны аналогичные элементы отображения и редактирования текстов, содержащихся в базах данных, так как их рассмотрение выходит за рамки данной книги.

Таблица 3.2. Компоненты ввода и отображения чисел, дат и времени

Пиктограмма	Компонент	Страница	Описание
	UpDown (кнопка-счетчик)	Win32	Кнопка-счетчик в стиле Windows 95, в сочетании с компонентами Edit и другими позволяющая вводить цифровую информацию. Основное свойство — Position .
	SpinEdit (кнопка-счетчик с окном редактирования)	Samples	Окно редактирования в комбинации с кнопкой-счетчиком. Почти то же, что комбинация Edit и UpDown . Основное свойство — Value .
	DateTimePicker (окно ввода дат и времени)	Win32	Ввод даты (с выпадающим календарем) и времени. Основные свойства — Date и Time .

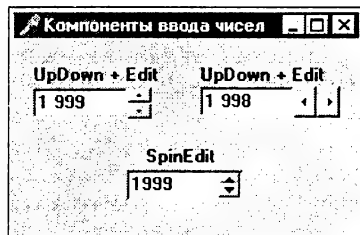
Пикто- грамма	Компонент	Страница	Описание
	MonthCalendar (окно ввода дат)	Win32	Ввод дат с выбором из календаря.
	Calendar (календарь на указанный месяц)	Samples	Отображение календаря на указанный месяц. Компонент DateTimePicker имеет больше возможностей по вводу дат, чем этот компонент. Основные свойства — Month и Day .
	F1Book (страницы Excel)	ActiveX	Компонент ввода и обработки числовой информации, аналогичный страницам Excel.

3.4.2 Ввод и отображение целых чисел — компоненты UpDown и SpinEdit

В Delphi имеются специализированные компоненты, обеспечивающие ввод целых чисел — **UpDown** и **SpinEdit** (см. пример на рис. 3.12).

Рис. 3.12

Пример компонентов UpDown и SpinEdit



Компонент **UpDown** превращает окно редактирования **Edit** в компонент, в котором пользователь может выбирать целое число, изменяя его кнопками со стрелками. Если к тому же установить в **true** свойство окна **ReadOnly**, то пользователь просто не сможет ввести в окно какой-либо свой текст и вынужден будет ограничиться выбором числа. Компонент **SpinEdit** представляет собой сочетание **Edit** и **UpDown**, оформленное как отдельный тип компонента.

Основное свойство компонента **UpDown** — **Associate**, связывающее кнопки со стрелками с одним из оконных компонентов, обычно с **Edit**. Чтобы опробовать компонент **UpDown**, перенесите на форму его и окно редактирования **Edit**, расположив **Edit** там, где это требуется, а **UpDown** — в любом месте формы. Далее в вы-

падающем списке свойства **Associate** компонента **UpDown** выберите **Edit1**. Компонент **UpDown** немедленно переместится к **Edit** и как бы сольется с ним.

Свойство **AlignButton** компонента **UpDown**, которое может принимать значения **udLeft** или **udRight**, определяет, слева или справа от окна будут размещаться кнопки. Свойство **Orientation**, которое может принимать значения **udHorizontal** или **udVertical**, определяет, расположатся ли кнопки по вертикали (одна под другой — см. левый компонент на рис. 3.12) или по горизонтали (одна рядом с другой — см. правый компонент на рис. 3.12). Свойство **ArrowKeys** определяет, будут ли управлять компонентом клавиши клавиатуры со стрелками. Свойство **Thousands** определяет наличие или отсутствие разделительного пробела между каждыми тремя цифрами разрядов вводимого числа.

Свойства **Min** и **Max** компонента **UpDown** задают соответственно минимальное и максимальное значения чисел, свойство **Increment** задает приращение числа при каждом нажатии на кнопку. Свойство **Position** определяет текущее значение числа. Это свойство можно читать, чтобы узнать, какое число ввел пользователь. Его можно задать во время проектирования в диапазоне **Min** — **Max**. Тогда это будет значение числа по умолчанию, отображаемое в окне в начале выполнения приложения.

Свойство **Wrap** определяет, как ведет себя компонент при достижении максимального или минимального значений. Если **Wrap = false**, то при увеличении или уменьшении числа до максимального или минимального значения это число фиксируется на предельном значении и нажатие кнопки, пытающейся увеличить максимальное число или уменьшить минимальное, ни к чему не приводит. Если же **Wrap = true**, то попытка превысить максимальное число приводит к его сбросу на минимальное значение. Аналогично, попытка уменьшить минимальное число приводит к его сбросу на максимальное значение. Т.е. изменение чисел «закольцовывается».

Если в компоненте **Edit**, связанном с **UpDown**, не задать **ReadOnly** равным **true**, то пользователь сможет редактировать число, не пользуясь кнопками со стрелками. Это удобно, если требуемое число далеко от указанного по умолчанию, а шаг приращения **Increment** в **UpDown** мал. Но тут проявляется серьезный недостаток компонента **UpDown**: ничто не мешает пользователю ввести по ошибке не цифры, а какие-то другие символы. Чтобы избавиться от этого недостатка, можно использовать прием, описанный в разделе 3.3.3, который не дает возможность пользователю ввес-

ти в окно редактирования какие-то символы, кроме цифр. Но лучше для этих целей использовать компонент **SpinEdit**.

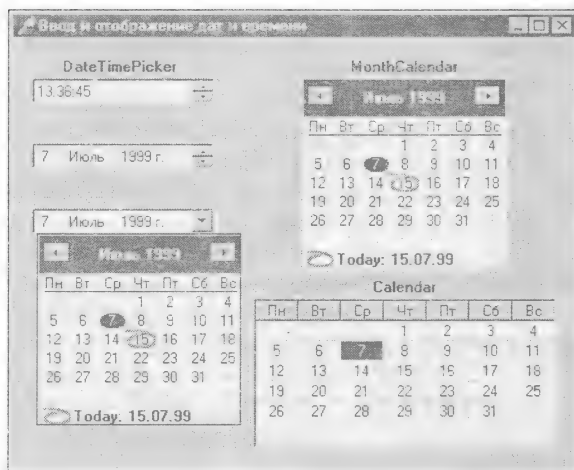
Свойства компонента **SpinEdit** похожи на рассмотренные, только имеют другие имена: свойства **Min**, **Max**, **Position** называются соответственно **MinValue**, **MaxValue**, **Value**. В целом компонент **SpinEdit** во многих отношениях удобнее простого сочетания **UpDown** и **Edit**. Так что, если не требуются какие-то из описанных выше дополнительных возможностей **UpDown** (нестандартное расположение кнопок, «закольцовывание» изменений и т.п.), то можно рекомендовать пользоваться компонентом **SpinEdit**.

3.4.3 Ввод и отображение дат и времени — компоненты **DateTimePicker**, **MonthCalendar**, **Calendar**

Примеры компонентов ввода и отображение дат и времени приведены на рис. 3.13.

Рис. 3.13

Примеры компонентов отображения дат и времени



Из этих компонентов наиболее удобным является **DateTimePicker** (на рис. 3.13, слева сверху показан этот компонент в режиме ввода времени, а ниже — в двух вариантах режима ввода даты). Компонент очень эффектен за счет появления выпадающего календаря (иногда даже слишком эффектен для строго оформленного приложения) и обеспечивает безошибочный с точки зрения синтаксиса ввод дат и времени. Его свойство **Kind** определяет режим работы компонента: **dtkDate** — ввод даты, **dtkTime** — ввод времени.

При вводе дат можно задать свойство **DateMode** равным **dmComboBox** — наличие выпадающего календаря, или равным

dmUpDown — наличие кнопок увеличения и уменьшения (см. средний компонент **DateTimePicker** на рис. 3.13), напоминающих те, которые используются в описанных ранее компонентах **UpDown** и **SpinEdit**. Только в данном случае пользователь может независимо устанавливать с помощью кнопок число, месяц и год. Формат представления дат определяется свойством **DateFormat**, которое может принимать значения **dfShort** — краткий формат (например, 01.12.99), или **dfLong** — полный формат (например, 1 Декабрь 1999г.).

Значение даты по умолчанию можно задать в Инспекторе Объектов через свойство **Date**. Это же свойство читается для определения заданной пользователем даты. При чтении **Date** надо учитывать тип этого свойства — **TDateTime**, представляющий собой число с плавающей запятой, целая часть которого содержит число дней, отсчитанное от некоторого начала календаря, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. Для 32-разрядных версий Delphi за начало календаря принята дата 12/30/1899 00 часов. В Delphi 1 за начало отсчета принят год 1, т.е. для перевода даты Delphi 1 в дату последующих версий Delphi надо вычесть из даты число 693594.

Для преобразования значения свойства **Date** в строку можно воспользоваться функцией **DateToStr**. Например, оператор

```
Memol.Lines.Add('Дата: ' +  
DateToStr(DateTimePicker1.Date));
```

добавит в окно **Memol** строку вида «Дата: 01.12.98».

При вводе дат можно задать значения свойств **MaxDate** и **MinDate**, определяющих соответственно максимальную и минимальную дату, которую может задать пользователь.

В режиме ввода времени **dtkTime** введенное пользователем значение можно найти в свойстве **Time**, тип которого — тот же рассмотренный выше **TDateTime**. Преобразовать время в строку можно функцией **TimeToStr**.

Компонент **MonthCalendar** похож на компонент **DateTimePicker**, работающий в режиме ввода дат. Правда, в компоненте **MonthCalendar** предусмотрены некоторые дополнительные возможности: можно допустить множественный выбор дат в некотором диапазоне (свойство **MultiSelect**), можно указывать в календаре номера недель с начала года (свойство **WeekNumbers**), перестраивать календарь, задавая первый день каждой недели (свойство **FirstDayOfWeek**) и т.п. Для некоторых офисных приложений все это достаточно удобно.

Компонент **Calendar** представляет собой менее красочный и более обыденно оформленный календарь на один месяц. Вместо

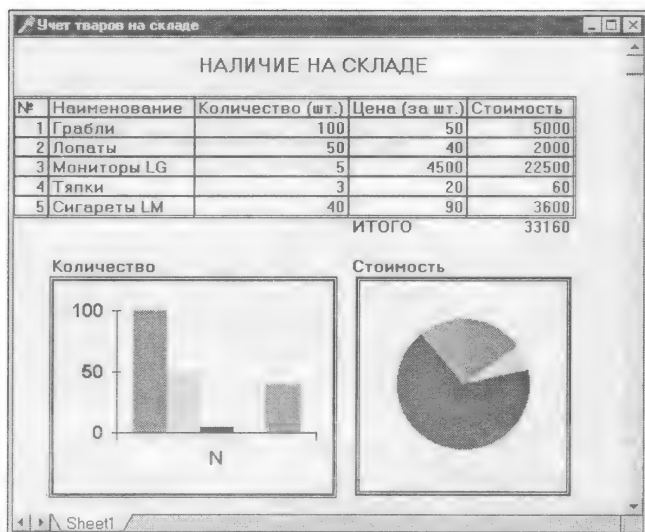
свойства **Date** в нем предусмотрены отдельные свойства **Year** — год, **Month** — месяц, **Day** — день. Все это целые числа, с которыми иногда удобнее иметь дело, чем с типом **TDateTime**. Перед отображением на экране или в процессе проектирования надо задать значения **Month** и **Year**, чтобы компонент отобразил календарь на указанный месяц указанного года. Впрочем, если вам надо иметь календарь на текущий месяц, надо установить в **true** значение свойства **UseCurrentDate** (установлено по умолчанию). В этом случае по умолчанию будет показан календарь на текущий месяц с выделенным в нем текущим днем. Свойство **StartOfWeek** задает день, с которого начинается неделя. По умолчанию задано 0 — воскресенье, как это принято в западных календарях. Но для нас все-таки как-то привычнее начинать неделю с рабочего дня — понедельника. Так что желательно задать **StartOfWeek = 1**.

3.4.4 Страницы Excel — компонент F1Book

Очень интересным компонентом является **F1Book** на странице **ActiveX**. Этот компонент позволяет встроить в ваше приложение таблицы типа **Excel** (рис. 3.14), которые пользователь может заполнять соответствующими числами, а компонент будет производить по заданным формулам вычисления и тут же отображать их результаты в указанных ячейках. В таблицу можно встроить диаграммы и графики различных типов. И все изменения, вносимые пользователем в данные таблицы, немедленно будут отобра-

Рис. 3.14

Приложение
с компонентом
F1Book

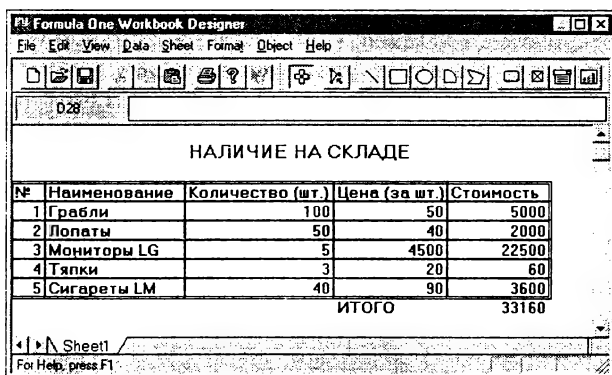


жаться в диаграммах. Таким образом вы можете включать в свое приложение различные бланки смет, счетов, ведомостей, с которыми будет работать пользователь, различные таблицы, производящие статистические или технические расчеты и т.п.

Перенесите на форму компонент **F1Book** и щелкните на нем правой кнопкой мыши. Выберите из всплывшего меню команду **Workbook Designer**. Перед вами появится диалоговое окно проектирования, представленное на рис. 3.15. Те, кто знаком с программой Excel, могут увидеть, что это окно является несколько упрощенным вариантом Excel. Проектирование таблицы производится фактически по тем же правилам, что и в Excel. Вы можете писать в ячейках необходимые надписи, задавая шрифт, его стиль, обрамление. Можете записывать формулы. Так на рис. 3.14 и 3.15 последний столбец представляет собой стоимость соответствующего товара, являющуюся произведением его количества на его цену. А ячейка внизу таблицы суммирует стоимость всех товаров.

Рис. 3.15

Диалоговое окно проектирования компонента F1Book



Правая быстрая кнопка на рис. 3.15 позволяет ввести на страницу диаграммы и графики. Чтобы задать диаграмму, надо сначала выделить курсором в таблице данные, которые должны отображаться в диаграмме, затем нажать кнопку ввода диаграммы, после этого указать курсором рамку, в которой должна отображаться диаграмма. В результате вы попадете в диалоговое окно, в котором сможете выбрать тип диаграммы и необходимые ее атрибуты.

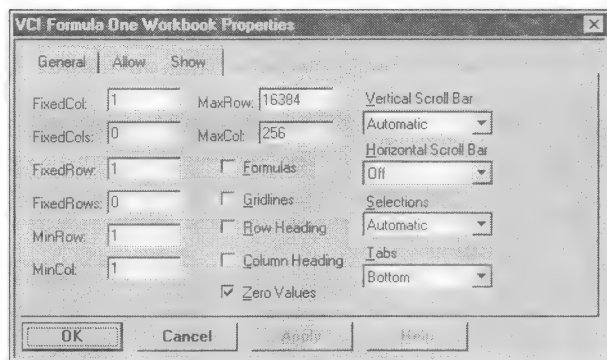
Рассказывать подробно о работе с окном проектирования компонента **F1Book** невозможно из-за ограничения на объем данной книги. Те, кто знаком с Excel, без труда смогут в этом окне ори-

ентироваться. К тому же в нем имеется встроенная справка, вызываемая командой меню Help или клавишей F1.

Щелкнув правой кнопкой мыши на компоненте **F1Book**, вы можете выбрать еще одну команду — Properties. В появившемся при этом диалоговом окне (рис. 3.16) вы можете, в частности, задать опции, определяющие, что будет видно или не видно в таблице при работе приложения: заголовки строк и столбцов (Row Heading и Column Heading), сетка (Gridlines), формулы вычислений (Formulas) и т.п.

Рис. 3.16

Диалоговое окно задания свойств объекта F1Book



3.5 Обеспечение синтаксически правильного ввода текстовых и цифровых данных

Мы рассмотрели различные компоненты ввода информации. Теперь остановимся на проблеме сокращения числа возможных ошибок пользователя при вводе текстовых и числовых данных. Это очень серьезная проблема, особенно при построении приложений, рассчитанных на массового и не очень квалифицированного пользователя. Частично безошибочного ввода можно добиться за счет маскирования, описанного при рассмотрении компонента **MaskEdit**. Однако, и при маскированном вводе пользователь может ошибиться в синтаксисе, в результате чего будет генерироваться исключение. Еще лучше использовать, если возможно, выбор с помощью компонентов типа **ListBox**, **ComboBox**, **StringGrid**, **DateTimePicker**, **SpinEdit** и т.п. Это гарантирует, если не правильный в смысловом плане, то, по крайней мере, синтаксически правильный ввод. В качестве примера на рис. 3.17 приведены два варианта ввода информации о сотрудниках некоей организации. Слева ввод осуществляется с помощью компонентов **Edit** и пользователь может сделать любые ошибки.

Справа ввод осуществляется с помощью компонентов **ComboBox**, **SpinEdit**, **MaskEdit** и **DateTimePicker**. **ComboBox** со значением свойства **Style** равным **csDropDownList** не допускает редактирования и, значит, пользователь просто не может указать неправильный отдел. **SpinEdit** гарантирует синтаксически правильное указание стажа работы, а его свойства **MinValue** и **MaxValue** определяют реальные пределы вводимого числа (например, число лет от 1 до 50). Аналогично компонент **MaskEdit** гарантирует синтаксически правильный ввод номера телефона, а компонент **DateTimePicker** гарантирует (даже без использования выпадающего календаря) синтаксически правильный ввод даты рождения и ее допустимые пределы (например, от 1930 г. до 1980г.).

Рис. 3.17

Пример правильной и неправильной организации ввода информации

Form1

Плохая организация ввода

Отдел

Бухгалтерия

Данные о сотруднике Иванове

Стаж

125

Телефон

A23-4567

Дата рождения

7.5.970

Правильная организация ввода

Отдел

Бухгалтерия

Бухгалтерия

Управление

Д. Цех 1

Цех 2

Стаж

10

Телефон

123-45-67

Дата рождения

7 Май 1970 г.

3.6 Компоненты отображения заголовков и иерархических данных

3.6.1 Перечень компонентов

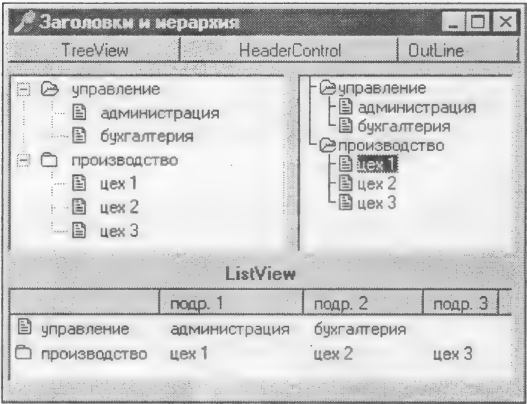
Ниже будут рассмотрены компоненты секционированного отображения текстов (заголовков) и компоненты отображения иерархических данных. Перечень этих компонентов дается в таблице 3.3.

Таблица 3.3. Компоненты отображения секционированных текстов и иерархических данных

Пиктограмма	Компонент	Страница	Описание
	HeaderControl (заголовков)	Win32	Создание составных перемещаемых заголовков в стиле Windows 95/98.
	Header (заголовков)	Win3.1	Создание составных перемещаемых заголовков. 16-битный вариант компонента HeaderControl .
	TreeView (окно дерева данных)	Win32	Просмотр структуры иерархических данных в стиле Windows 95/98.
	List View (список данных в стиле Windows 95)	Win32	Отображение в стиле Windows 95/98 списков в колонках или в виде пиктограмм.
	Outline (окно дерева данных)	Win3.1	Просмотр структуры иерархических данных в стиле Windows 3.x.

На рис. 3.18 приведены примеры перечисленных в таблице компонентов.

Рис. 3.18
Примеры компонентов заголовков и отображения иерархических данных



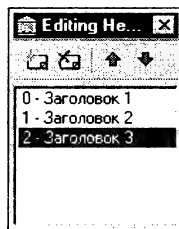
3.6.2 Секционированное отображение текстов — компоненты **HeaderControl** и **Header**

Компоненты заголовков **HeaderControl** и **Header** можно рассматривать в какой-то степени как частный случай таблицы строк, состоящей всего из одной строки. По умолчанию свойство **Align** в **HeaderControl** задано равным **alTop**, что обеспечивает размещение компонента вверху окна формы. Но это свойство можно изменить, например, на **alNone** и разместить компонент в любом необходимом месте. Заголовок состоит из ряда секций, причем пользователь во время выполнения приложения может изменять ширину отдельных секций с помощью мыши.

Основное свойство компонента **HeaderControl** — **Sections**. Оно является списком объектов типа **THeaderSection**, каждый из которых описывает одну секцию заголовка. Свойство **Sections** можно задать во время проектирования, нажав кнопку с многоточием рядом с этим свойством в Инспекторе Объектов или просто сделав двойной щелчок на компоненте **HeaderControl**. В обоих случаях перед вами откроется окно редактора заголовков, представленное на рис. 3.19. Левая быстрая кнопка позволяет добавить новую секцию в заголовок. Следующая быстрая кнопка позволяет удалить секцию. Кнопки со стрелкой позволяют изменять последовательность секций.

Рис. 3.19

Окно редактора заголовков



После того, как вы добавили секцию и установили на ней курсор, в окне Инспектора Объектов появится множество свойств этого объекта. В свойстве **Text** вы можете задать текст заголовка. Свойства **MinWidth** и **MaxWidth** определяют соответственно минимальную и максимальную ширину секции в пикселях. Только в этих пределах пользователь может изменять ширину секции курсором мыши. Значение ширины по умолчанию задается значением свойства **Width**. При изменении ширины секции во время выполнения генерируется событие **OnSectionResize**. В обработчик этого события надо вставить операторы, синхронно изменяющие ширину того, заголовком чего является секция: это может быть столбец какой-то таблицы, какая-то панель и т.п.

Свойство **AllowClick**, равное по умолчанию **true**, определяет поведение секции как кнопки при щелчке пользователя на ней. В этом случае при щелчке генерируется событие **OnSectionClick** компонента **HeaderControl**, в обработчик которого и надо вставить операторы, выполняющие необходимые действия.

Свойство **Style** может иметь значение **hsText** — в этом случае в заголовке отображается значение свойства **Text**, или **hsOwnerDraw** — в этом случае отображается то, что рисуется непосредственно на канве операторами, записанными в обработчике события **OnDrawSection** компонента **HeaderControl**.

Компонент **Header** обладает существенно меньшими возможностями, чем **HeaderControl**. В нем свойство **Sections** имеет тип **TStrings** и содержит только тексты заголовков, не позволяя регулировать пределы изменения ширины секций, их функционирование как кнопок и т.д. Таким образом, **Header** имеет смысл использовать только в Delphi 1.

3.6.3 Компоненты отображения иерархических данных — **TreeView**, **Outline**, **ListView**

Компоненты **TreeView** и **Outline** служат для отображения иерархических данных в виде дерева (см. пример на рис. 3.18), в котором пользователь может выбрать нужный ему узел или узлы. Иерархическая информация может быть самой разной: структура некоторого предприятия, структура документации учреждения, структура отчета и т.п. С каждым узлом дерева могут быть связаны некоторые данные.

Возможности компонента **TreeView** несколько шире, чем компонента **Outline**. К тому же **TreeView** — 32-разрядный компонент, а **Outline** — 16-разрядный. Поэтому **Outline** целесообразно использовать только в приложениях, создаваемых с помощью Delphi 1.

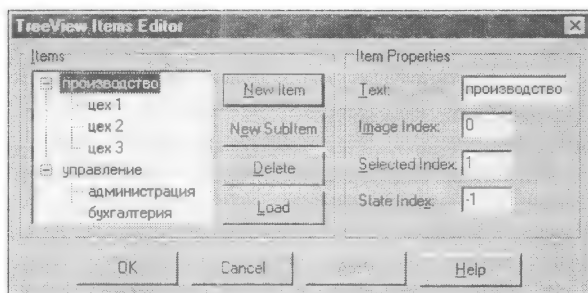
Основным свойством **TreeView**, содержащим информацию об узлах дерева, является **Items**. Доступ к информации об отдельных узлах осуществляется через этот индексный список узлов. Например, **TreeView1.Items[1]** — это узел дерева с индексом 1 (второй узел дерева). Каждый узел является объектом типа **TTreeNode**, обладающим своими свойствами и методами.

Во время проектирования формирование дерева осуществляется в окне редактора узлов дерева, представленном на рис. 3.20. Это окно вызывается двойным щелчком на компоненте **TreeView** или нажатием кнопки с многоточием около свойства **Items** в окне Инспектора Объектов.

Кнопка **New Item** (новый узел) позволяет добавить в дерево новый узел. Он будет расположен на том же уровне, на котором рас-

Рис. 3.20

Окно редактора узлов
дерева компонента
TreeView



положен узел, выделенный курсором в момент щелчка на кнопке New Item.

Кнопка New SubItem (новый дочерний узел) позволяет добавить в дерево дочерний узел. Он будет расположен на уровень ниже уровня того узла, который выделен курсором в момент щелчка на кнопке New SubItem.

Кнопка Delete (удалить) удаляет выделенный узел дерева. Кнопка Load позволяет загрузить структуру дерева из файла. Файл, хранящий структуру дерева — это обычный текстовый файл, содержащий тексты узлов. Уровни узлов обозначаются отступами. Например, файл дерева, изображенного на рис. 3.18 и 3.20, может иметь вид:

```
производство
  цех 1
  цех 2
  цех 3
управление
  администрация
  бухгалтерия
```

Для каждого нового узла дерева можно указать ряд свойств в панели Item Properties окна на рис. 3.20. Это прежде всего свойство Text — надпись, появляющаяся в дереве около данного узла. Свойства Image Index и Selected Index определяют индекс пиктограммы, отображаемой для узла, который соответственно не выделен и выделен пользователем в данный момент. Эти индексы соответствуют списку изображений, хранящихся в отдельном компоненте ImageList (см. раздел 9.3). Указание на этот компонент вы можете задать в свойстве Images компонента TreeView. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), пиктограммы изображаться не будут. Последнее свойство — State Index в панели Item Properties позволяет добавить вторую пиктограмму в данный узел, не зависящую от состояния узла. Подобная пиктограмма может просто служить дополните-

льной характеристикой узла. Индекс, указываемый как `State Index`, соответствует списку изображений, хранящихся в отдельном компоненте `ImageList`, указанном в свойстве `StateImages` компонента `TreeView`.

Мы рассмотрели формирование дерева в процессе проектирования. Однако, дерево можно формировать или перестраивать и во время выполнения приложения. Для этого служит ряд методов объектов типа `TTreeNode`. Следующие методы позволяют вставлять в дерево новые узлы:

function Add(Node: TTreeNode; const S: string): TTreeNode;	Добавляет новый узел с текстом S как последний узел уровня, на котором расположен Node .
function AddFirst(Node: TTreeNode; const S: string): TTreeNode;	Вставляет новый узел с текстом S как первый из узлов уровня, на котором находится Node . Индексы последующих узлов увеличиваются на 1.
function Insert(Node: TTreeNode; const S: string): TTreeNode;	Вставляет новый узел с текстом S сразу после узла Node на то же уровень. Индексы последующих узлов увеличиваются на 1.
function AddChild(Node: TTreeNode; const S: string): TTreeNode;	Добавляет узел с текстом S как последний дочерний узла Node .
function AddChildFirst(Node: TTreeNode; const S: string): TTreeNode;	Вставляет новый узел с текстом S как первый из дочерних узлов узла Node . Индексы последующих узлов увеличиваются на 1.

Каждый из этих методов возвращает вставленный узел.

Ниже в качестве примера приведен код, формирующий то же дерево, которое вы можете видеть на рис. 3.18 и 3.20.

```
TreeView1.Items.Clear; // очистка списка
// добавление корневого узла 'производство' (индекс 0)
TreeView1.Items.Add(nil, 'производство');

{добавление дочерних узлов 'цех 1' — 'цех 3' (индексы 1
— 3)}
TreeView1.Items.AddChild(TreeView1.Items.Item[0],
                         'цех 1');
TreeView1.Items.AddChild(TreeView1.Items.Item[0],
                         'цех 2');
TreeView1.Items.AddChild(TreeView1.Items.Item[0],
                         'цех 3');
```

```
{ добавление корневого узла 'управление' после узла
  'производство' (индекс 4) }
TreeView1.Items.Add(TreeView1.Items.Item[0],
  'управление');
```

```
{ добавление дочерних узлов 'администрация' и
  'бухгалтерия' узла 'управление' }
TreeView1.Items.AddChild(TreeView1.Items.Item[4],
  'администрация');
TreeView1.Items.AddChild(TreeView1.Items.Item[4],
  'бухгалтерия');
```

Дерево может быть сколь угодно разветвленным. Например, следующие операторы добавляют дочерние узлы «бригада 1» и «бригада 2» в сформированный ранее узел «цех 1»:

```
TreeView1.Items.AddChild(TreeView1.Items.Item[1],
  'бригада 1');
TreeView1.Items.AddChild(TreeView1.Items.Item[1],
  'бригада 2');
```

Текст, связанный с некоторым узлом, можно найти с помощью его свойства **Text**. Например, **TreeView1.Items.Item[1].Text** — это надпись «цех 1».

С каждым узлом может быть связан некоторый объект. Добавление таких узлов осуществляется методами **AddObject**, **AddObjectFirst**, **InsertObject**, **AddChildObject**, **AddChildObjectFirst**, аналогичными приведенным выше, но содержащими в качестве параметра еще указатель на объект:

```
function AddObject(Node:  
  TTreeNode; const S: string;  
  Ptr: Pointer): TTreeNode;
```

Добавляет новый узел с текстом **S** и объектом **Ptr** как последний узел уровня, на котором расположен **Node**.

```
function AddObjectFirst(Node:  
  TTreeNode; const S: string;  
  Ptr: Pointer): TTreeNode;
```

Вставляет новый узел с текстом **S** и объектом **Ptr** как первый из узлов уровня, на котором находится **Node**. Индексы последующих узлов увеличиваются на 1.

```
function InsertObject(Node:  
  TTreeNode; const S: string;  
  Ptr: Pointer): TTreeNode;
```

Вставляет новый узел с текстом **S** и объектом **Ptr** сразу после узла **Node** на то же уровень. Индексы последующих узлов увеличиваются на 1.

```
function AddChildObject(Node:  
  TTreeNode; const S: string;  
  Ptr: Pointer): TTreeNode;
```

Добавляет узел с текстом **S** и объектом **Ptr** как последний дочерний узла **Node**.

function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;	Вставляет новый узел с текстом S и объектом Ptr как первый из дочерних узлов узла Node . Индексы последующих узлов увеличиваются на 1.
---	---

Объект, связанный с некоторым узлом, можно найти с помощью его свойства **Data**. Например, **TreeView1.Items.Item[1].Data**.

Для удаления узлов имеется два метода: **Clear**, очищающий все дерево, и **Delete(Node: TTreeNode)**, удаляющий указанный узел **Node** и все его узлы — потомки. Например, оператор

```
TreeView1.Items.Clear;
```

удалит в нашем примере все узлы, а оператор

```
TreeView1.Items.Delete(TreeView1.Items.Item[1]);
```

удалит узел «цех 1» и его дочерние узлы (если они имеются).

При удалении узлов, связанных с объектами, сами эти объекты не удаляются.

Реорганизация дерева, связанная с созданием или удалением многих узлов, может вызывать неприятное мерцание изображения. Избежать этого можно с помощью методов **BeginUpdate** и **EndUpdate**. Первый из них запрещает перерисовку дерева, а второй — разрешает. Таким образом, изменение структуры дерева может осуществляться по следующей схеме:

```
TreeView1.Items.BeginUpdate;  
<операторы изменения дерева>  
TreeView1.Items.EndUpdate;
```

Если метод **BeginUpdate** применен подряд несколько раз, то перерисовка дерева произойдет только после того, как столько же раз будет применен метод **EndUpdate**.

Среди свойств узлов следует отметить **Count** — число узлов, управляемых данным, т.е. дочерних узлов, их дочерних узлов и т.п. Если значение **Count** узла равно нулю, значит у узла нет дочерних узлов, т.е. он является листом дерева.

Вернемся к свойствам компонента **TreeView**. Важным свойством компонента **TreeView** является **Selected**. Это свойство указывает узел, который выделен пользователем. Пользуясь этим свойством можно запрограммировать операции, которые надо выполнить для выбранного пользователем узла. Если ни один узел не выбран, значение **Selected** равно **nil**. При выделении пользователем нового узла происходят события **OnChanging** (перед изменением выделения). В обработчик события передаются параметры **Node: TTreeNode** — узел, который выделен в данный момент, и **var AllowChange: Boolean** — разрешение на перенос вы-

деления. Если в обработчике задать **AllowChange = false**, то переключение выделения не произойдет.

У компонента **TreeView** имеется свойство **RightClickSelect**, разрешающее (при значении равно **true**) выделение узла щелчком как левой, так и правой кнопкой мыши.

Ряд событий компонента **TreeView** связан с развертыванием и свертыванием узлов. При развертывании узла происходят события **OnExpanding** (перед развертыванием) и **OnExpanded** (после развертывания). В обработчики обоих событий передается параметр **Node: TTreeNode** — развертываемый узел. Кроме того в обработчик **OnExpanding** передается параметр **var AllowExpansion: Boolean**, который можно задать равным **false**, если желательно запретить развертывание. При свертывании узла происходят события **OnCollapsing** (перед свертыванием) и **OnCollapsed** (после свертывания). Так же, как и в событиях, связанных с развертыванием, в обработчики передается параметр **Node: TTreeNode** — свертываемый узел, а в обработчик **OnCollapsing** дополнительно передается параметр **var AllowCollapse: Boolean**, разрешающий или запрещающий свертывание.

Свойство **ReadOnly** компонента **TreeView** позволяет запретить пользователю редактировать отображаемые данные. Если редактирование разрешено, то при редактировании возникают события **OnEditing** и **OnEdited**, аналогичные рассмотренным ранее (в обработчике **OnEditing** параметр **var AllowEdit: Boolean** позволяет запретить редактирование).

Ряд свойств компонента **TreeView: ShowButtons, ShowLines, ShowRoot** отвечают за изображение дерева и позволяют отображать или убирать из него кнопки, показывающие раскрытия узла, линии, связывающие узлы, и корневой узел. Поэкспериментируйте с этими свойствами и увидите, как они влияют на изображение.

Свойство **SortType** позволяет автоматически сортировать ветви и узлы дерева. По умолчанию это свойство равно **stNone**, что означает, что дерево не сортируется. Если установить **SortType** равным **stText**, то узлы будут автоматически сортироваться по алфавиту. Возможно также проводить сортировку по связанным с узлами объектам **Data** (значение **SortType** равно **stData**), одновременно по тексту и объектам **Data** (значение **SortType** равно **stBoth**) или любым иным способом. Для использования этих возможностей сортировки надо написать обработчик события **OnCompare**, в который передаются, в частности, параметры **Node1** и **Node2** — сравниваемые узлы, и по ссылке передается целый параметр **Compare**, в который надо заносить результат сравне-

ния: отрицательное число, если узел **Node1** должен располагаться ранее **Node2**, 0, если эти узлы считаются эквивалентными, и положительное число, если узел **Node1** должен располагаться в дереве после **Node2**. Например, следующий оператор в обработчике события **OnCompare** обеспечивает обратный алфавитный порядок расположения узлов:

```
Compare := - AnsiCompareText(Node1.Text, Node2.Text);
```

События **OnCompare** наступают после задания любого значения **SortType**, отличного от **stNone**, и при изменении пользователем свойств узла (например, при редактировании им надписи узла), если значение **SortType** не равно **stNone**. После сортировки первоначальная последовательность узлов в дереве теряется. Поэтому последующее задание **SortType = stNone** не восстанавливает начальное расположение узлов, но исключает дальнейшую генерацию событий **OnCompare**, т.е. автоматическую перестановку узлов, например, при редактировании их надписей. Если же требуется изменить характер сортировки или провести сортировку с учетом новых созданных узлов, то надо сначала задать значение **SortType = stNone**, а затем задать любое значение **SortType**, отличное от **stNone**. При этом будут сгенерированы новые обращения к обработчику событий **OnCompare**.

Имеются и другие возможности сортировки. Например, метод **AlphaSort** обеспечивает алфавитную последовательность узлов независимо от значения **SortType**, но при отсутствии обработчика событий **OnCompare** (если обработчик есть, то при выполнении метода **AlphaSort** происходит обращение к этому обработчику). Отличие метода **AlphaSort** от задания значения **SortType = stText** заключается в том, что изменение надписей узлов приводит к автоматической пересортировке дерева только при **SortType = stText**.

Мы рассмотрели основные возможности компонента **TreeView**. Компонент **Outline** похож на него. Структура дерева тоже содержится в свойстве **Items** и доступ к отдельным узлам также осуществляется через этот индексный список узлов. Но индексы начинаются с 1. Например, **Outline1.Items[1]** — это узел дерева с индексом 1 (первый узел дерева). Свойство **Items** имеет тип **TOutlineNode**. Его свойства и методы отличаются от свойств и методов типа узлов в **TreeView**. И заполняется структура дерева иначе: через свойство **Lines** типа **TStrings**. Редактор этого свойства можно вызвать, щелкнув на кнопке с многоточием около свойства **Lines** в окне Инспектора Объектов. Вы попадете в окно, в котором можете записать тексты всех узлов, делая отступы, чтобы

выделить уровни узлов. Текст должен выглядеть так, как выше описывалось представление структуры в текстовом файле.

Пиктограммы, сопровождающие изображения узлов, задаются параметрами **PictureOpen** (пиктограмма развернутого узла), **PictureClosed** (пиктограмма свернутого узла), **PictureMinus** (пиктограмма символа «-» около развернутого узла, имеющего наследников), **PicturePlus** (пиктограмма символа «+» узла, имеющего наследников, но не развернутого), **PictureLeaf** (пиктограмма узла, не имеющего наследников — листа дерева). Основное отличие пиктограмм компонента **Outline** заключается в том, что они одинаковы для всех узлов одного типа, тогда как в **TreeView** можно задавать пиктограммы индивидуально для каждого узла.

Программно изменять структуру дерева можно с помощью методов **Add**, **AddObject** (добавление узла в дерево), **Insert**, **InsertObject** (вставка узла в заданную позицию), **AddChild**, **AddChildObject** (вставка дочернего узла), **Delete** (удаление узла).

Индекс выделенного пользователем узла можно определить через свойство **SelectedItem**. Если **SelectedItem = 0**, значит ни один узел не выделен. Текст выделенного узла определяется свойством **Text**: например,

```
OutLine1.Items[OutLine1.SelectedItem].Text
```

Впрочем, тот же самый текст даст и выражение

```
OutLine1.Lines[OutLine1.SelectedItem-1]
```

В заключение коротко рассмотрим компонент **ListView**. Он позволяет отображать в стиле Windows 95/98 данные в виде списков, таблиц, крупных и мелких пиктограмм. С подобным отображением все вы сталкиваетесь, раскрывая папки Windows.

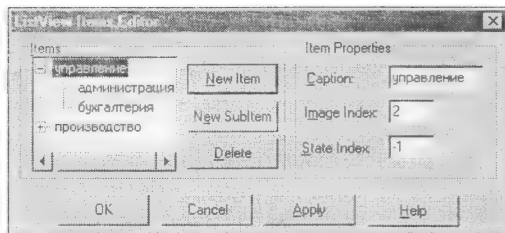
Стиль отображения информации определяется свойством **ViewStyle**, которое может устанавливаться в процессе проектирования или программно во время выполнения. Свойство может принимать значения: **vsIcon** — крупные значки, **vsSmallIcon** — мелкие значки, **vsList** — список, **vsReport** — таблица. Что означает каждое из этих значений вы можете посмотреть в любой папке Windows на рабочем столе.

Если вы хотите предоставить пользователю возможность изменять вид списка, сделайте следующее:

- введите в приложение разделы меню Крупные значки (пусть его имя будет **MIcon**), Мелкие значки (имя **MSmallIcon**), Список (имя **MList**) и Таблица (имя **MReport**)
- установите во всех этих разделах одинаковый отличный от нуля индекс **GroupIndex** и свойства **RadioItem** в **true**

Рис. 3.21

Окно редактора списка объектов компонента ListView



- один из разделов пометьте как **Checked** и в свойстве списка **ViewStyle** установите значение, соответствующее этому разделу
- напишите следующие обработчики щелчков для этих разделов:

```
procedure TForm1.MIconClick(Sender: TObject);
begin
    ListView1.ViewStyle:=vsIcon;
    MIcon.Checked:=true;
end;
```

```
procedure TForm1.MSmallIconClick(Sender: TObject);
begin
    ListView1.ViewStyle:=vsSmallIcon;
    MSmallIcon.Checked:=true;
end;
```

```
procedure TForm1.MListClick(Sender: TObject);
begin
    ListView1.ViewStyle:=vsList;
    MList.Checked:=true;
end;
```

```
procedure TForm1.MReportClick(Sender: TObject);
begin
    ListView1.ViewStyle:=vsReport;
    MReport.Checked:=true;
end;
```

Основное свойство компонента, описывающее состав отображаемой информации — **Items**. Во время проектирования оно может быть установлено специальным редактором (рис. 3.21), вызываемом щелчком на кнопке с многоточием рядом с этим свойством в окне Инспектора Объектов. Оно похоже на окно, описанное для компонента **TreeView** (рис. 3.20). Точно так же в нем задаются новые узлы кнопкой **New Item** и дочерние узлы — кнопкой **New SubItem**. Только смысл дочерних узлов другой: это информация, которая появляется только в режиме **vsReport** — в виде таблицы.

Для каждого узла задается свойство **Caption** — надпись, появляющаяся около пиктограммы. Для дочерних узлов это свойство соответствует надписи, появляющейся в ячейках таблицы в режиме **vsReport**.

Свойство **Image Index** определяет индекс пиктограммы. Индекс соответствует спискам изображений, хранящимся в отдельных компонентах **ImageList** (см. раздел 9.3). Указания на эти компоненты вы можете задать в свойствах **LargeImages** для режима **vsIcon** и **SmallImages** для режимов **vsSmallIcon**, **vsList** и **vsReport**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), пиктограммы изображаться не будут. Свойство **State Index** в панели **Item Properties** позволяет добавить вторую пиктограмму в данный объект. Подобная пиктограмма может просто служить дополнительной характеристикой объекта. Индекс, указываемый как **State Index**, соответствует списку изображений, хранящихся в отдельном компоненте **ImageList**, указанном в свойстве **StateImages** компонента **ListView**.

Свойство **Columns** определяет список заголовков таблицы в режиме **vsReport** при свойстве **ShowColumnHeaders** (показать заголовки), установленном в **true**. Свойство **Columns** можно задать в процессе проектирования специальным редактором заголовков, вызываемом двойным щелчком на компоненте **ListView** или щелчком на кнопке с многоточием рядом со свойством **Columns** в окне Инспектора Объектов. Редактор заголовков уже был нами рассмотрен (см. рис. 3.19).

Свойство **Checkboxes** определяет отображение индикатора с флажком около каждого элемента списка. Индикаторы можно устанавливать программно или их может изменять пользователь во время выполнения. Тогда узнать программно, установлен ли индикатор в некотором элементе **Items[i]**, можно проверкой его свойства **Checked**. Например:

```
for i:=0 to ListView1.Items.Count-1 do  
  if ListView1.Items[i].Checked  
  then ...;
```

Свойства **HotTrack** и **HotTrackStyles** определяют появление выделения при перемещении курсора над элементом списка и стиль этого выделения. Свойство **HoverTime** (только в Delphi 5) задает в миллисекундах задержку появления такого выделения.

Свойство списка **Selected** определяет выделенный пользователем элемент списка.

Свойство **DragMode** определяет режим перетаскивания элементов списка. Если вы хотите разрешить пользователю перетас-

кивать пиктограммы по площади списка в режимах **vsIcon** и **vsSmallIcon**, вам надо сделать следующее:

- установить **DragMode** = **dmAutomatic**
- установить **DragKind** = **dkDrag**
- написать обработчик события **OnDragOver** списка:
`Accept := Source = ListView1;`
- написать обработчик события **OnDragDrop** списка:
`(Source as TListView).Selected.Position:=Point(X,Y);`
- если вы предоставили пользователю возможность выбирать вид отображения списка с помощью описанных ранее разделов меню, то в обработчиках щелчка на **MIcon** и **MSmallIcon** надо написать операторы
`ListView1.DragMode:=dmAutomatic;`

а в обработчиках щелчка на **MList** и **MReport** надо написать операторы

```
ListView1.DragMode:=dmManual;
```

Метод **Arrange**:

```
procedure Arrange(Code: TListArrangement);
```

позволяет упорядочить пиктограммы в режимах **vsIcon** и **vsSmallIcon**. Параметр **Code** определяет способ упорядочивания:

arAlignBottom	выравнивание вдоль нижнего края области
arAlignLeft	выравнивание вдоль левого края области
arAlignRight	выравнивание вдоль правого края области
arAlignTop	выравнивание вдоль верхнего края области
arDefault	выравнивание по умолчанию (вдоль верхнего края области)
arSnapToGrid	размещение каждой пиктограммы в ближайшем узле сетки

В Delphi 5 введено еще одно свойство **ListView** — **WorkAreas**. Это свойство определяет рабочую область (прямоугольную рамку), в которой осуществляется выравнивание пиктограмм в режимах **vsIcon** и **vsSmallIcon**. Свойство **WorkAreas** представляет собой индексированный список, аналогичный **Items**, но совершенно независимый от него. Если **WorkAreas** — пустой список (ни одна область в него не добавлена), то упорядочивание пиктограмм производится в пределах всей клиентской области **ListView**. Добавление новой рабочей области осуществляется методом **Add**. Свойство рабочей области **Rect** типа **TRect** (см. раздел 3.2) опреде-

ляет границы области. Свойство **Color** определяет цвет рамки, обрамляющей область. Свойство **DisplayName** определяет подпись под рамкой. И рамка, и подпись отображаются только в случае, если свойство списка **ShowWorkAreas** установлено в **true**.

Пример операторов, создающих в площади списка две рабочие области:

```
ListView1.WorkAreas.Add;  
ListView1.WorkAreas[0].Rect:=Rect(0,0,  
                                   ListView1.ClientWidth div 4,  
                                   ListView1.ClientHeight div 2);  
ListView1.WorkAreas.Items[0].DisplayName:='Область 0';  
ListView1.WorkAreas.Items[0].Color := clRed;  
ListView1.WorkAreas.Add;  
ListView1.WorkAreas[1].Rect:=Rect(  
                                   ListView1.ClientWidth div 2,0,  
                                   (ListView1.ClientWidth div 4)*3,  
                                   ListView1.ClientHeight div 2);  
ListView1.WorkAreas.Items[1].DisplayName:='Область 1';
```

Упорядочивание пиктограмм происходит в пределах той области, в которой они находятся. Если вы разрешили пользователю перетаскивать пиктограммы, как было описано ранее, то вид выравнивания будет зависеть от расположения пиктограмм. Если одна или несколько из них расположены в пределах одной области, а другие размещаются вне рабочих областей, то при вызове метода **Arrange** все они расположатся в пределах той области, в которой была хоть одна пиктограмма. Если же пиктограммы были расположены в нескольких рабочих областях, то они будут упорядочиваться в пределах областей их размещения.

Способ упорядочивания определяется соответствующим заданием свойства **SortType**, которое уже рассматривалось нами для компонента **TreeView**.

4.1 Перечень компонентов отображения графической информации

Для отображения графической информации в библиотеке Delphi предусмотрены компоненты, список которых дан в таблице 4.1.

Таблица 4.1. Компоненты отображения графической информации

Пиктограмма	Компонент	Страница	Описание
	Image (изображение)	Additional	Используется для отображения графики: пиктограмм, битовых матриц и метафайлов.
	PaintBox (окно для рисования)	System	Используется для создания на форме некоторой области, в которой можно рисовать.
	DrawGrid (таблица рисунков)	Additional	Используется для отображения в строках и столбцах текстовых данных.
	Chart (диаграммы и графики)	Additional	Компонент принадлежит к семейству компонентов TChart , которые используются для создания диаграмм и графиков.

Кроме того, отображать и вводить графическую информацию можно на поверхности любого оконного компонента, имеющего свойство **Canvas** — канва.

4.2 Отображение графики на канве Canvas

Канва **Canvas** не является компонентом, так что, строго говоря, она не должна бы рассматриваться в рамках данной книги. Но поскольку многие компоненты, в частности, формы, имеют канву и канва предоставляет возможность выводить различную графическую информацию, то некоторые начальные сведения о канве все-таки целесообразно дать.

Канва представляет собой область компонента, на которой можно рисовать или отображать готовые изображения. Она содержит свойства и методы, существенно упрощающие графику Delphi. Все сложные взаимодействия с системой спрятаны для пользователя, так что рисовать в Delphi может человек, совершенно не искушенный в машинной графике.

Каждая точка канвы имеет координаты **X** и **Y**. Система координат канвы, как и везде в Delphi, имеет началом левый верхний угол канвы. Координата **X** возрастает при перемещении слева направо, а координата **Y** — при перемещении сверху вниз. Координаты измеряются в пикселях. *Пиксель* — это наименьший элемент поверхности рисунка, с которым можно манипулировать. Важнейшее свойство пикселя — его цвет.

Канва имеет свойство **Pixels**. Это свойство представляет собой двумерный массив, который отвечает за цвета канвы. Например, **Canvas.Pixels[10,20]** соответствует цвету пикселя, 10-го слева и 20-го сверху. С массивом пикселей можно обращаться как с любым свойством: изменять цвет, задавая пикселю новое значение, или определять его цвет по хранящемуся в нем значению. Например, **Canvas.Pixels[10,20] := 0** или **Canvas.Pixels[10,20] := clBlack** — это задание пикселю черного цвета.

Свойство **Pixels** можно использовать для рисования на канве. Давайте попробуем нарисовать по пикселям график синусоиды на канве формы. Для этого в обработчик события формы **OnPaint** (прорисовка) можно вставить следующий код:

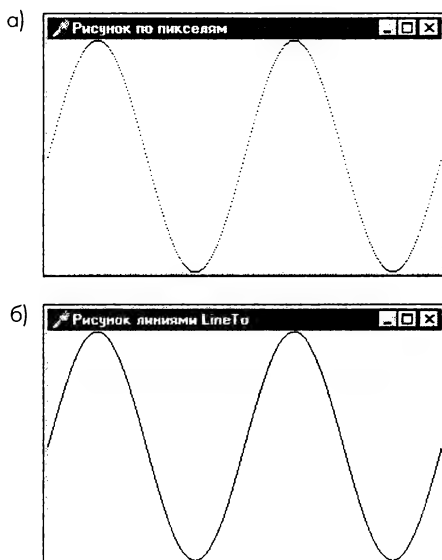
```
procedure TForm1.FormCreate(Sender: TObject);
var
  X,Y:real;           // координаты функции
  PX,PY:longint;      // координаты пикселей
begin
  Color:=clWhite;
  for PX:=0 to ClientWidth do
    begin
      {X — аргумент графика, соответствующий пикселю
      с координатой PX}
      X:= PX*4*Pi/ClientWidth;
      Y:=Sin(X);
      {PY — координата пикселя, соответствующая координате Y}
      PY:=trunc(ClientHeight-(Y+1)*ClientHeight/2);
      {Устанавливается черный цвет выбранного пикселя
      (0 яркости)}
      Canvas.Pixels[PX,PY] := 0;
    end;
  end;
```

Выполните это тестовое приложение и вы увидите результат, представленный на рис. 4.1 а. График синусоиды получился,

хотя и не очень хороший, т.к. разбивается на отдельные точки — пиксели.

Рис. 4.1

График синусоиды, построенный по пикселям (а) и линиями (б)



Канва — объект класса **TCanvas** имеет множество методов, которые позволяют рисовать графики, линии, фигуры с помощью свойства **Pen** — перо. Это свойство является объектом, в свою очередь имеющим ряд свойств. Одно из них уже известное вам свойство **Color** — цвет, которым наносится рисунок. Второе свойство — **Width** (ширина линии). Ширина задается в пикселях. По умолчанию ширина равна 1.

Свойство **Style** определяет вид линии. Это свойство может принимать следующие значения:

psSolid	Сплошная линия
psDash	Штриховая линия
psDot	Пунктирная линия
psDashDot	Штрих-пунктирная линия
psDashDotDot	Линия, чередующая штрих и два пунктира
psClear	Отсутствие линии
psInsideFrame	Сплошная линия, но при Width > 1 допускающая цвета, отличные от палитры Windows

У канвы имеется свойство **PenPos** типа **TPoint**. Это свойство определяет в координатах канвы текущую позицию пера. Перемещение пера без прорисовки линии, т.е. изменение **PenPos**, производится методом канвы **MoveTo(X,Y)**. Здесь (X, Y) — координаты точки, в которую перемещается перо. Эта текущая точка становится исходной, от которой методом **LineTo(X,Y)** можно провести линию в точку с координатами (X,Y). При этом текущая точка перемещается в конечную точку линии и новый вызов **LineTo** будет проводить точку из этой новой текущей точки.

Давайте попробуем нарисовать пером график синуса из предыдущего примера. В данном случае обработчик события формы **OnPaint** может иметь вид:

```
procedure TForm1.FormPaint(Sender: TObject);
var
  X,Y:real;           // координаты функции
  PX,PY:longint;      // координаты пикселей
begin
  Color:=clWhite;
  Canvas.MoveTo(0,ClientHeight div 2);
  for PX:=0 to ClientWidth do
    begin
    {X — аргумент графика, соответствующий пикселю с
    координатой PX}
      X:= PX*4*Pi/ClientWidth;
      Y:=Sin(X);
    {PY — координата пикселя, соответствующая координате Y}
      PY:=trunc(ClientHeight-(Y+1)*ClientHeight/2);
    {Проводится линия на графике}
      Canvas.LineTo(PX,PY);
    end;
  end;
```

Результат работы приложения в этом варианте вы можете видеть на рис. 4.1 б. Как видите, качество графика существенно улучшилось.

Перо может рисовать не только прямые линии, но и фигуры. Полный список методов канвы, использующих перо, см. во встроенной справке Delphi. А пока в качестве примера приведем только один из них — **Ellipse**, который рисует эллипс или окружность. Он объявлен как

```
procedure Ellipse(X1, Y1, X2, Y2: Integer);
```

где параметры **X1, X2, Y1, Y2** определяют координаты прямоугольника, описывающего эллипс или окружность. Например, оператор

```
Canvas.Ellipse(10,40,20,50);
```

нарисует окружность с диаметром 10 и с координатами центра (15, 45).

Фигуры в общем случае рисуются не пустыми, а закрашенными с помощью свойства канвы **Brush** — кисть. Свойство **Brush** является объектом, имеющим в свою очередь ряд свойств. Свойство **Color** определяет цвет заполнения. Свойство **Style** определяет шаблон заполнения (штриховку). По умолчанию значение **Style** равно **bsSolid**, что означает сплошное закрашивание цветом **Color**.

У пера **Pen** имеется еще одно свойство, которое мы пока не рассматривали. Это свойство — **Mode** (режим). По умолчанию значение **Mode** = **pmCopy**. Это означает, что линии проводятся цветом, заданным в свойстве **Color**. Но возможны и другие режимы, в которых учитывается не только цвет **Color**, но и цвет соответствующих пикселей фона. Наиболее интересным из этих режимов является режим **pmNotXor** — сложение с фоном по инверсному исключающему ИЛИ. Если задан этот режим, то повторное рисование той же фигуры на том же месте канвы убирает ранее нарисованное изображение и восстанавливает цвета пикселей, которые были до первого изображения фигуры.

Эту особенность режима **pmNotXor** можно использовать для создания простенькой анимации. Достаточно нарисовать нечто, затем стереть нарисованное, перерисовать немного измененным — и рисунок будет представляться ожившим.

Попробуйте сделать сами простенькую мультипликацию — движущуюся окружность. Начните новое приложение и в раздел **implementation** вставьте объявление

```
var X,Y:integer;
```

Тем самым вы введете глобальные переменные **X** и **Y** — текущие координаты изображения.

В событие формы **OnPaint** вставьте операторы

```
Canvas.Brush.Color:=clWhite;  
Color:=clWhite;  
Canvas.Pen.Mode:=pmNotXor;
```

Первый из этих операторов задает белый цвет кисти **Brush**. Значит ваша окружность будет закрашена внутри белым цветом. Второй оператор задает белый цвет фона поверхности формы. Третий оператор устанавливает режим пера **pmNotXor**, который позволит вам стирать прежние изображения прежде, чем нарисовать новое.

Даже самая простая мультипликация нуждается в синхронизации. Иначе скорость движения будет определяться быстродействием компьютера. Поэтому перенесите на форму компонент **Ti-**

mer — таймер со страницы **System**. Этот компонент описан в разделе 5.7. Можете посмотреть там его подробное описание. А пока задайте его свойство **Interval** равным, например, 30 (это время выдержки в миллисекундах, но реальное время выдержки будет больше — см. раздел 5.7) и установите свойство **Enabled** равным **false** (это означает, что таймер не будет запускаться автоматически в момент запуска приложения).

В обработчик события этого компонента **OnTimer** вставьте операторы

```
// Стирание прежнего изображения
Canvas.Ellipse(X-5,Y,X+5,Y-10);
Inc(X);
// Рисование нового изображения
Canvas.Ellipse(X-5,Y,X+5,Y-10);
// Останов при достижении конца формы
if(X>=ClientWidth-20) then Timer1.Enabled:=false;
```

Первый из этих операторов рисует окружность в том месте, где она была нарисована ранее, т.е. стирает прежнее изображение. Далее увеличивается на единицу функцией **Inc** текущая координата **X** и изображение окружности рисуется в новой позиции. Последний оператор останавливает изображение у края формы.

Теперь перенесите на форму кнопку **Button** и в обработчик щелчка на ней поместите операторы

```
X:=10;
Y:=100;
Canvas.Ellipse(X-5,Y,X+5,Y-10);
Timer1.Enabled:=true;
```

Первые два оператора задают начальные координаты окружности. Третий оператор рисует окружность в ее начальном положении, а четвертый — запускает таймер.

Оттранслируйте приложение, запустите его на выполнение, щелкните на кнопке. Вы увидите изображение окружности, перемещающееся по форме слева направо. А дальше уж подключите вашу фантазию и преобразуйте это не слишком интересное приложение во что-нибудь более увлекательное.

На канве можно отображать не только программно создаваемые изображения, но и изображения, хранящиеся в графических файлах. Только сама канва не имеет метода загрузки изображения из файла. Поэтому загружать файл надо в какой-нибудь другой графический объект, способный воспринимать информацию графических файлов. А затем переписывать изображение из этого объекта на канву с помощью метода канвы **Draw**. Его описание:

```
procedure Draw(X, Y: Integer; Graphic: TGraphic);
```

Здесь параметры **X** и **Y** определяют координаты левого верхнего угла размещения изображения на канве, а **Graphic** — объект, хранящий информацию. В качестве такого объекта может выступать, например, объект типа **TBitmap**, предназначенный для хранения битовых матриц. Давайте посмотрим, как все это выглядит на практике.

Откройте новое приложение, перенесите на форму компонент **OpenPictureDialog** со страницы Dialogs (это компонент диалога открытия графических файлов — см. раздел 8.2) и кнопку **Button**. Разместите **OpenPictureDialog** в любом месте формы, так как этот компонент не визуальный, а кнопку разместите внизу формы. В обработчик щелчка на кнопке занесите код:

```
procedure TForm1.Button1Click(Sender: TObject);
var BitMap:TBitmap;
begin
// Выбор пользователем графического файла
if OpenPictureDialog1.Execute
then begin
// Создание объекта BitMap типа TBitmap
BitMap:=TBitmap.Create;
// Загрузка в BitMap выбранного графического файла
BitMap.LoadFromFile(OpenPictureDialog1.FileName);
// Перенос изображения на канву формы
Canvas.Draw(10,10,BitMap);
// Уничтожение объекта BitMap
BitMap.Free;
end;
end;
```

Этот код создает временный объект типа **TBitmap** с именем **BitMap**. Затем вызывается диалог открытия графического файла **OpenPictureDialog1** и, если пользователь выбрал файл, то он загружается в **BitMap** методом **LoadFromFile**. Затем методом **Draw** загруженное изображение копируется на канву в область, с координатами левого верхнего угла (10, 10). После этого временный объект **BitMap** уничтожается.

Запустите ваше приложение и щелкните на его кнопке. Вы увидите, что можете загрузить любой графический файл типа **.bmp** и он отобразится на канве формы (см. рис. 4.2 а). Графические файлы вы можете найти в каталоге Images. В Delphi 5 и 4 он обычно расположен в каталоге ...\\program files\\Common Files\\Borland Shared. В Delphi 3 он расположен в каталоге ...\\program files\\Borland\\Delphi 3, а в Delphi 1 — в каталоге Delphi16. В каталоге Images имеется, в частности, подкаталог \\Images\\Splash\\16Color\\, в котором хранится файл, загруженный в примере рис. 4.2.

Рис. 4.2

Изображение на канве графического файла (а) и его стирания (б) при перекрытии другим окном



Вы создали неплохое приложение для просмотра графических файлов. Но теперь давайте попробуем увидеть его крупный недостаток. Не закрывая своего приложения, перейдите в какую-нибудь другую программу, например, вернитесь в Delphi. Затем, ничего там не делая, опять перейдите в свое выполняющееся приложение. Если окно программы, в которую вы уходили, целиком перекрыло окно вашего приложения, то вернувшись в него вы увидите, что картинка в окне исчезла. Если же окно вашего приложения перекрывалось только частично, то вернувшись в свое приложение вы, возможно, увидите результат, подобный представленному на рис. 4.2 б.

Вы видите, что если окно какого-то другого приложения перекрывает на время окно вашего приложения, то изображение, нарисованное на канве формы, портится. Посмотрим, как можно устранить этот недостаток.

Если окно было перекрыто и изображение испортилось, операционная система сообщает приложению, что в окружении что-то изменилось и что приложение должно предпринять соот-

ветствующие действия. Как только требуется обновление окна, для него генерируется событие **OnPaint**. В обработчике этого события (в нашем случае события формы) нужно перерисовать изображение.

Перерисовка может производиться разными способами в зависимости от приложения. В нашем примере можно было бы вынести объявление переменной **BitMap** (оператор **var BitMap:TBitMap**) за пределы приведенной выше процедуры, т.е. сделать эту переменную глобальной, разместив непосредственно в разделе **implementation**. Оператор **BitMap.Free** можно было бы перенести в обработчик события формы **OnDestroy**, происходящего в момент закрытия приложения. Тогда в течение всего времени выполнения вашего приложения вы будете иметь копию картинки в компоненте **BitMap** и вам достаточно ввести в обработчик события **OnPaint** формы всего один оператор:

```
Canvas.Draw(10,10,BitMap);
```

Сделайте это, и увидите, что изображение на форме не портится при любых перекрытиях окон.

Помимо рассмотренного метода **Draw** канва имеет еще метод копирования **CopyRect**:

```
procedure CopyRect(Dest:TRect; Canvas:TCanvas;  
                   Source:TRect);
```

Метод копирует указанную параметром **Source** область изображения в канве источника изображения **Canvas** в указанную параметром **Dest** область данной канвы. Тип **TRect**, характеризующий прямоугольные области **Source** и **Dest**, уже описывался в разделе 3.2. Например, оператор

```
Canvas.CopyRect(MyRect2,Bitmap.Canvas,MyRect1);
```

копирует на канву формы в область **MyRect2** изображение из области **MyRect1** канвы компонента **Bitmap**.

Копирование методом **CopyRect** производится в режиме, установленном свойством **CopyMode**. По умолчанию это свойство имеет значение **CmSrcCopy**, что означает просто замену изображения, содержащегося ранее в области **Dest**, на копируемое изображение. Другие возможные значения **CopyMode** позволяют комбинировать изображения, но их рассмотрение выходит за рамки данной книги.

Этими основными сведениями о выводе графической информации на канву мы ограничимся. В разделе 3.2 были сообщены сведения о выводе на канву текста. В целом же канва — сложный объект, обладающий еще многими свойствами и методами. Но это требует развернутого обсуждения, выходящего за рамки

данной книги. В следующей книге серии «Все о Delphi» эти вопросы будут рассмотрены подробнее.

4.3 Компоненты Image и PaintBox

Компоненты **Image** и **PaintBox** представляют собой некоторую ограниченную поверхность с канвой, на которую можно заносить изображения, как это описано в разделе 4.2. При этом компонент **PaintBox**, собственно говоря, не дает ничего нового по сравнению с рисованием на канве формы. Рисование на **PaintBox** вместо формы не имеет никаких преимуществ, кроме, может быть, некоторого облегчения в расположении одного или нескольких рисунков в площади окна.

Компонент **Image** много богаче по своим возможностям и удобнее, чем **PaintBox**. Даже при использовании описанных в разделе 4.2 приемов рисования на канве компонент **Image** имеет существенное преимущество: в нем не приходится думать о перерисовке изображения, испорченного из-за перекрытия данного окна другими. Все, связанное с обработкой событий **OnPaint**, рассмотренных в разделе 4.2, в **Image** осуществляется автоматически. Кроме того с помощью **Image** проще, чем при непосредственном рисовании на канве формы, расположить в окне приложения несколько изображений и управлять ими. При этом отпадает необходимость сложных и нудных расчетов координат канвы формы, обеспечивающих требуемое взаимное расположение изображений, т.е. в полной мере проявляются преимущества визуального программирования. Так что, вероятно, во всех случаях лучше работать с канвой **Image**, чем с канвой формы.

Но помимо этих возможностей у компонента **Image** имеются свойства, позволяющие работать с различными типами графических файлов.

Delphi поддерживает три типа файлов — битовые матрицы, пиктограммы и метафайлы. Все три типа файлов хранят изображения; различие заключается лишь в способе их хранения внутри файлов и в средствах доступа к ним. Битовая матрица (файл с расширением **.bmp**) отображает цвет каждого пикселя в изображении. При этом информация хранится таким образом, что любой компьютер может отобразить изображение с разрешающей способностью и количеством цветов, соответствующими его конфигурации.

Пиктограммы (файлы с расширением **.ico**) — это маленькие битовые матрицы. Они повсеместно используются для обозначения значков приложений, в быстрых кнопках, в пунктах меню, в

различных списках. Способ хранения изображений в пиктограммах схож с хранением информации в битовых матрицах, но имеются и различия. В частности, пиктограмму невозможно масштабировать, она сохраняет тот размер, в котором была создана.

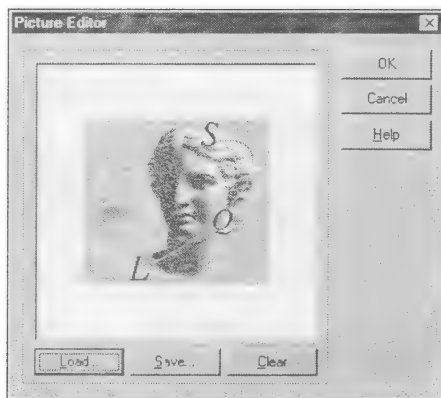
Метафайлы (Metafiles) хранят не последовательность битов, из которых состоит изображение, а информацию о способе создания картинки. Они хранят последовательности команд рисования, которые и могут быть повторены при воссоздании изображения. Это делает такие файлы, как правило, более компактными, чем битовые матрицы.

Компонент **Image** позволяет отображать информацию, содержащуюся в графических файлах всех указанных типов. Для этого служит его свойство **Picture** — объект типа **TPicture**.

Чтобы познакомиться с этим свойством откройте новое приложение и перенесите на форму компонент **Image**. Растяните его или задайте его свойство **Align** равным **alClient**, чтобы он занял всю клиентскую область формы. Нажмите на кнопку с многоточием около свойства **Picture** в окне Инспектора Объектов или просто сделайте двойной щелчок на **Image**. Перед вами откроется окно **Picture Editor** (рис. 4.3), позволяющее загрузить в свойство **Picture** какой-нибудь графический файл (кнопка **Load**), а также сохранить открытый файл под новым именем или в новом каталоге. Щелкните на **Load**, чтобы загрузить графический файл. Перед вами откроется окно открытия графического файла, представленное на рис. 4.4. По мере перемещения курсора в списке по графическим файлам в правом окне отображаются содержащиеся в них картинки, а над ними — цифры, характеризующие размер картинки. Вы можете выбрать требуемый вам графический файл любого типа. Напомним, что поставляемые с Delphi

Рис. 4.3

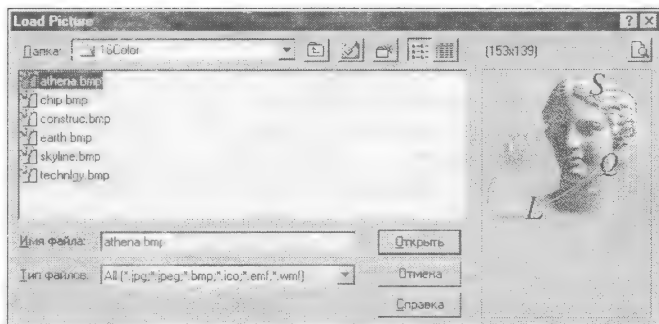
Окно **Picture Editor**



графические файлы вы можете найти в каталоге Images. В Delphi 5 и 4 он обычно расположен в каталоге ...\\program files\\Common Files\\Borland Shared. В Delphi 3 он расположен в каталоге ...\\program files\\Borland\\Delphi 3, а в Delphi 1 — в каталоге Delphi16. После загрузки файла щелкните на ОК в окне Picture Editor и в вашем компоненте **Image** отобразится выбранная вами картинка. Можете запустить ваше приложение и полюбоваться ею. Впрочем, вы и так видите картинку, даже не выполняя приложение.

Рис. 4.4

Окно диалога
открытия
графического
файла



Когда вы в процессе проектирования загрузили изображение из файла в компонент **Image**, он не просто отображает его, но и сохраняет в приложении. Это дает вам возможность поставлять ваше приложение без отдельного графического файла. Впрочем, как мы увидим позднее, в **Image** можно загружать и внешние графические файлы в процессе выполнения приложения.

Вернемся к рассмотрению свойств компонента **Image**.

Если установить свойство **AutoSize** в **true**, то размер компонента **Image** будет автоматически подгоняться под размер помещенной в него картинки. Если же свойство **AutoSize** установлено в **false**, то изображение может не поместиться в компонент или, наоборот, площадь компонента может оказаться много больше площади изображения.

Другое свойство — **Stretch** позволяет подгонять не компонент под размер рисунка, а рисунок под размер компонента. Установите **AutoSize** в **false**, растяните или сожмите размер компонента **Image** и установите **Stretch** в **true**. Вы увидите, что рисунок займет всю площадь компонента, но поскольку вряд ли реально установить размеры **Image** точно пропорциональными размеру рисунка, то изображение исказится. Устанавливать **Stretch** в **true** может иметь смысл только для каких-то узоров, но не для

картинок. Свойство **Stretch** не действует на изображения пиктограмм, которые не могут изменять своих размеров.

Свойство — **Center**, установленное в **true**, центрирует изображение на площади **Image**, если размер компонента больше размера рисунка.

Рассмотрим еще одно свойство — **Transparent** (прозрачность). Если **Transparent** равно **true**, то изображение в **Image** становится прозрачным. Это можно использовать для наложения изображений друг на друга. Поместите на форму второй компонент **Image** и загрузите в него другую картинку. Только постарайтесь взять какую-нибудь мало заполненную, контурную картинку. Можете, например, взять картинку из числа помещаемых обычно на кнопки, например, стрелку (файл ...\\program files\\common files\\borland shared\\images\\buttons\\arrow1r.bmp). Передвиньте ваши **Image** так, чтобы они перекрывали друг друга, и в верхнем компоненте установите **Transparent** равным **true**. Вы увидите, что верхняя картинка перестала заслонять нижнюю. Одно из возможных применений этого свойства — наложение на картинку надписей, выполненных в виде битовой матрицы. Эти надписи можно сделать с помощью встроенной в Delphi программы **Image Editor**.

Учтите, что свойство **Transparent** действует только на битовые матрицы. При этом прозрачным (т.е. заменяемым на цвет расположенного под ним изображения) делается по умолчанию цвет левого нижнего пикселя битовой матрицы.

Мы рассмотрели загрузку изображения из файла в процессе проектирования. Но свойство **Picture** позволяет также легко организовать обмен с графическими файлами любых типов в процессе выполнения приложения. Чтоб пояснить технику такого обмена, надо сначала подробнее рассмотреть свойство **Picture**.

Это свойство является объектом, который имеет в свою очередь подсвойства, указывающие на хранящийся графический объект. Если в **Picture** хранится битовая матрица, на нее указывает свойство **Picture.Bitmap**. Если хранится пиктограмма, на нее указывает свойство **Picture.Icon**. На хранящийся метафайл указывает свойство **Picture.Metafile**. Наконец, на графический объект произвольного типа указывает свойство **Picture.Graphic**.

Объект **Picture** и его свойства **Bitmap**, **Icon**, **Metafile** и **Graphic** имеют методы файлового чтения и записи **LoadFromFile** и **SaveToFile**:

```
procedure LoadFromFile(const FileName: string);  
procedure SaveToFile(const FileName: string);
```

Для свойств **Picture.Bitmap**, **Picture.Icon** и **Picture.Metafile** формат файла должен соответствовать классу объекта: битовой матрице, пиктограмме, метафайлу. При чтении файла в свойство **Picture.Graphic** файл должен иметь формат метафайла. А для самого объекта **Picture** методы чтения и записи автоматически подстраиваются под тип файла. Поясним это на примере.

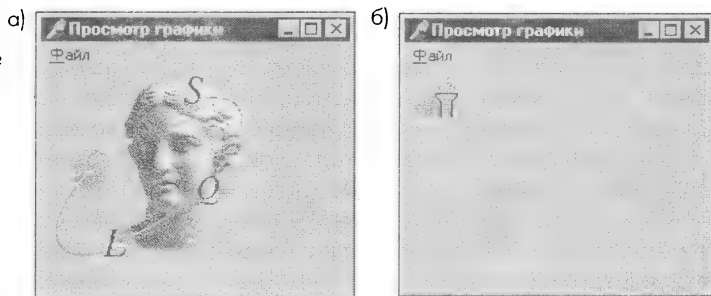
Давайте построим приложение, аналогичное рассмотренному в разделе 4.2 примеру просмотра графических файлов. Для разнообразия можно организовать управление им не кнопкой **Button**, а меню. Поместите на форму компонент **Image**. Растяните его или задайте его свойство **Align** равным **alClient**, чтобы он занял всю клиентскую область формы. Перенесите на форму компонент диалога открытия графического файла **OpenPictureDialog** (см. раздел 8.2). Поместите также на форму компонент главного меню **MainMenu** (см. раздел 6.1) и задайте в нем один раздел — Файл. В обработчике этого раздела напишите оператор

```
if (OpenPictureDialog1.Execute) then  
    Image1.Picture.LoadFromFile(  
        OpenPictureDialog1.FileName);
```

Этот оператор вызовет диалог открытия графического файла (см. рис. 4.4) и загрузит в компонент **Image1** изображение из выбранного пользователем файла (см. рис. 4.5). Причем файл может быть любого типа: битовая матрица, пиктограмма или метафайл.

Рис. 4.5

Изображение в компоненте **Image** битовой матрицы (а) и пиктограммы (б)



В этом приложении метод **LoadFromFile** применен к **Image1.Picture**. Если будут открываться только файлы битовых матриц, то оператор загрузки файла можно заменить на

```
Image1.Picture.Bitmap.LoadFromFile(  
    OpenPictureDialog1.FileName);
```

Для пиктограмм можно было бы использовать оператор

```
Image1.Picture.Icon.LoadFromFile(  
    OpenPictureDialog1.FileName);
```

а для метафайлов — оператор

```
Image1.Picture.Metafile.LoadFromFile(  
    OpenPictureDialog1.FileName);
```

или

```
Image1.Picture.Graphic.LoadFromFile(  
    OpenPictureDialog1.FileName);
```

Но во всех этих случаях, если формат файла не совпадет с предполагаемым, возникнет ошибка.

Аналогично работает и метод **SaveToFile** с тем отличием, что примененный к **Picture** или к **Picture.Graphic** он сохраняет в файле изображение любого формата. Например, если вы дополните свое приложение диалогом **SavePictureDialog** (см. раздел 8.2), введете в меню раздел Сохранить как и в его обработчик поместите оператор

```
if SavePictureDialog1.Execute then  
    Image1.Picture.SaveToFile(SavePictureDialog1.FileName);
```

то пользователь получит возможность сохранить изображение любого формата в файле с новым именем. Только при этом, чтобы не возникало в дальнейшем путаницы, расширение сохраняемого файла все-таки должно соответствовать формату сохраняемого изображения.

Абсолютно идентично для изображений любого формата будет работать программа, если оператор сохранения вы замените на

```
Image1.Picture.Graphic.SaveToFile(  
    SavePictureDialog1.FileName);
```

использующий свойство **Picture.Graphic**. А если вам известен формат хранимого в компоненте **Image** изображения, то вы можете применить метод **SaveToFile** к свойствам **Picture.Bitmap**, **Picture.Icon**, и **Picture.Metafile**.

Для всех рассмотренных объектов **Picture**, **Picture.Bitmap**, **Picture.Icon**, и **Picture.Metafile** определены методы присваивания значений объектов:

```
procedure Assign(Source: TPersistent);
```

Однако, для **BitMap**, **Icon** и **Metafile** присваивать можно только значения однородных объектов: соответственно битовых матриц, пиктограмм, метафайлов. При попытке присвоить значения разнородных объектов генерируется исключение **EConvertError**. Объект **Picture** — универсальный, ему можно присваивать значения объектов любых из остальных трех классов. А значение **Pic**

ture можно присваивать только тому объекту, тип которого совпадает с типом объекта, хранящегося в нем.

Метод **Assign** можно использовать и для обмена изображением с буфером **Clipboard**. Например, оператор

```
Clipboard.Assign (Image1.Picture);
```

занесет в буфер обмена изображение, хранящееся в **Image1**. Аналогично оператор

```
Image1.Picture.Assign (Clipboard);
```

прочитает в **Image1** изображение, находящееся в буфере обмена. Причем это может быть любое изображение и даже текст.

Надо только не забыть при работе с буфером обмена вставить в оператор **uses** вашего модуля ссылку на модуль **Clipbrd**. Автоматически Delphi эту ссылку не вставляет.

Возвращаясь к свойствам компонента **Image**, можно отметить один недостаток, присущий нашему тестовому приложению, приведенному на рис. 4.5. При загрузке разных изображений размер окна приложения может оказаться или слишком маленьким, и тогда вы увидите только часть изображения, или слишком большим, и тогда изображение будет некрасиво размещено в левом верхнем углу формы, оставляя много пустого пространства. Этот недостаток можно устранить, если воспользоваться свойствами **Height** (высота) и **Width** (ширина) компонента **Image**. При свойстве **AutoSize** установленном в **true** размеры **Image** автоматически устанавливаются равными размерам загруженного изображения. И этими размерами можно воспользоваться для соответствующего изменения размеров формы. Например, приведенный ранее код загрузки изображения из файла можно заменить на следующий:

```
if OpenPictureDialog1.Execute then
begin
  Image1.Picture.LoadFromFile(
    OpenPictureDialog1.FileName);
  Form1.ClientHeight := Image1.Height+10;
  Image1.Top:= Form1.ClientRect.Top
    +(Form1.ClientHeight-Image1.Height) div 2;
  Form1.ClientWidth := Image1.Width+10;
  Image1.Left:= Form1.ClientRect.Left
    +(Form1.ClientWidth-Image1.Width) div 2;
end;
```

В этом коде размеры клиентской области формы устанавливаются несколько больше размеров компонента **Image1**, которые в свою очередь адаптируются к размеру картинки благодаря свойству **AutoSize**. Внесите эти исправления в свое приложение, вы-

полните его и увидите, что форма стала автоматически адаптироваться к размерам загруженного изображения.

4.4 Таблицы изображений — компоненты **DrawGrid** и **StringGrid**

Компонент **DrawGrid** используется для создания в приложении таблицы, которая может содержать графические изображения (см. пример на рис. 4.6). Этот компонент подобен компоненту **StringGrid** (см. раздел 3.3.6), поскольку последний является производным от **DrawGrid**. Поэтому в **DrawGrid** присутствуют все свойства, методы, события компонента **StringGrid**, кроме относящихся к тексту, т.е. кроме свойств **Cells**, **Cols**, **Rows**, **Objects**. С этой точки зрения компонент **StringGrid** обладает существенно большими возможностями, чем **DrawGrid**, поскольку он может хранить в ячейках и изображения, и тексты. А если вы захотите внести текст в какие-то ячейки **DrawGrid**, то вам надо будет использовать для этого методы вывода текста на канву (см. раздел 3.2), что не очень удобно.

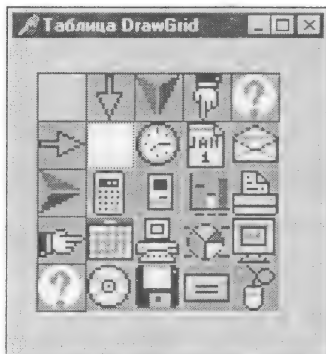
Рассмотрим свойства компонентов **DrawGrid** и **StringGrid**, относящиеся к изображениям, поскольку свойства **StringGrid**, относящиеся к тексту, уже рассматривались в разделе 3.3.6.

Компоненты **DrawGrid** и **StringGrid** имеют канву **Canvas**, на которой можно размещать изображения методами, изложенными в разделе 4.2. Имеется метод **CellRect**, который возвращает область канвы, отведенную под заданную ячейку. Этот метод определен как

```
function CellRect(ACol, ARow: Longint): TRect;
```

Рис. 4.6

Пример таблицы **DrawGrid**



где **ACol** и **ARow** — индексы столбца и строки, начинающиеся с 0, на пересечении которых расположена ячейка. Возвращаемая этой функцией область является областью канвы, в которой можно рисовать необходимое изображение. Например, оператор

```
DrawGrid1.Canvas.CopyRect(DrawGrid1.CellRect(1,1),
    Bitmap.Canvas, Rect(0,0,Bitmap.Height,Bitmap.Width));
```

копирует методом **CopyRect** (см. раздел 4.2) в ячейку (1,1) таблицы **DrawGrid1** изображение из компонента **Bitmap**. Эта ячейка является второй слева и второй сверху в таблице, поскольку индексы начинаются с 0. Учтите, что если размеры ячейки меньше, чем размер копируемого изображения, то в ячейке появится только левая верхняя часть картинка.

Изображение на канве компонентов **DrawGrid** и **StringGrid**, как и на канве любого компонента, подвержено стиранию при перекрытии окна приложения другими окнами или, например, при сворачивании приложения. Поэтому необходимо принимать меры, описанные в разделе 4.2, чтобы с помощью обработчика событий **OnPaint** восстанавливать испорченное изображение. Это делает компонент **DrawGrid** не слишком удобным для использования.

Все свойства и события, позволяющие определить выбранную пользователем ячейку таблицы, были рассмотрены в разделе 3.3.6. Там же вы найдете описание свойств, отвечающих за внешний вид и допустимость перестройки пользователем таблицы во время выполнения приложения.

4.5 Компонент Shape

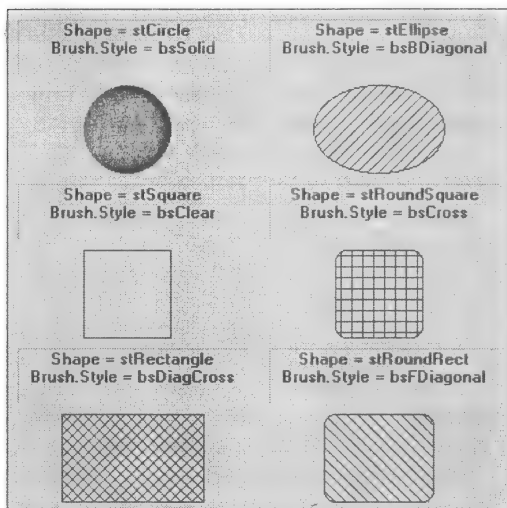
Компонент **Shape** только условно может быть отнесен к средствам отображения графической информации, поскольку просто представляет собой различные геометрические фигуры, соответствующим образом заштрихованные. Основное свойство этого компонента — **Shape** (форма), которое может принимать значения:

stRectangle	прямоугольник	stSquare	квадрат
stRoundRect	прямоугольник со скругленными углами	stRoundSquare	квадрат со скругленными углами
stEllipse	эллипс	stCircle	круг

Примеры этих форм показаны на рис. 4.7.

Рис. 4.7

Примеры компонента Shape



Другое существенное свойство компонента — **Brush** (кисть). Это свойство является объектом типа **TBrush**, имеющим ряд под-свойств, в частности: цвет (**Brush.Color**) и стиль (**Brush.Style**) заливки фигуры. Заливку при некоторых значениях **Style** вы можете видеть на рис. 4.7. Третье из специфических свойство компонента **Shape** — **Pen** (перо), определяющее стиль линий. Это свойство, как и свойство **Brush**, уже рассматривались в разделе 4.2. Справочные данные об этих свойствах вы можете найти в главе 10.

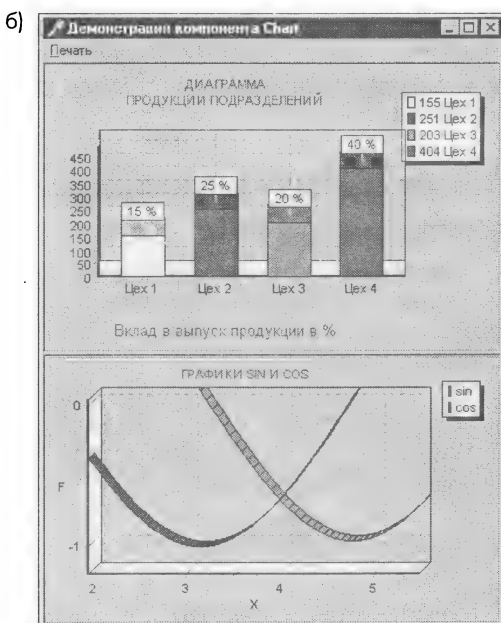
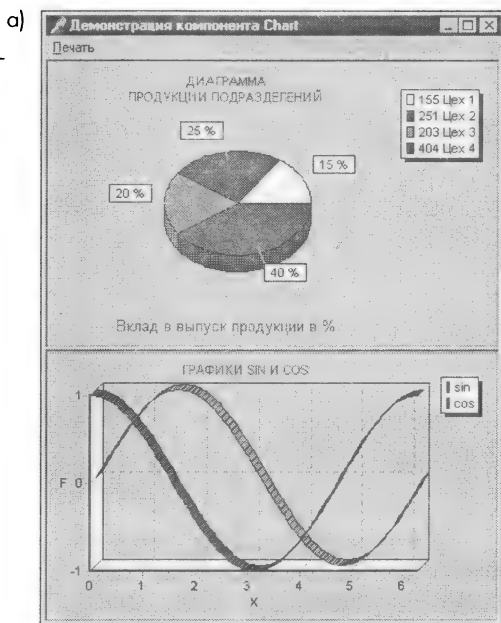
4.6 Построение графиков и диаграмм

4.6.1 Компонент Chart

Теперь рассмотрим компонент **Chart**. Этот компонент позволяет строить различные диаграммы и графики, которые выглядят очень эффектно (рис. 4.8). Компонент **Chart** имеет множество свойств, методов, событий, так что если все их рассматривать, то этому пришлось бы посвятить целую главу. Поэтому ограничимся рассмотрением только основных характеристик **Chart**. А с остальными вы можете ознакомиться во встроенной справке Delphi или просто опробовать их, экспериментируя с диаграммами.

Рис. 4.8

Пример приложения с диаграммами: начальное состояние (а) и состояние при изменении типа диаграммы и увеличении фрагмента графика (б)



Компонент **Chart** является контейнером объектов **Series** типа **TChartSeries** — серий данных, характеризующихся различными стилями отображения. Каждый компонент может включать несколько серий. Если вы хотите отображать график, то каждая серия будет соответствовать одной кривой на графике. Если вы хотите отображать диаграммы, то для некоторых видов диаграмм можно наложить друг на друга несколько различных серий, для других (например, для круговых диаграмм) это, вероятно, будет выглядеть некрасиво. Однако, и в этом случае вы можете задать для одного компонента **Chart** несколько серий одинаковых данных с разным типом диаграммы. Тогда, делая в каждый момент времени активной одну из них, вы можете предоставить пользователю выбор типа диаграммы, отображающей интересующие его данные.

Разместите один или два (если захотите воспроизвести рис. 4.8) компонента **Chart** на форме и посмотрите открывшиеся в Инспекторе Объектов свойства. Приведем пояснения некоторых из них.

AllowPanning

Определяет возможность пользователя прокручивать наблюдаемую часть графика во время выполнения, нажимая правую кнопку мыши. Возможные значения: **pmNone** — прокрутка запрещена, **pmHorizontal**, **pmVertical** или **pmBoth** — разрешена соответственно прокрутка только в горизонтальном направлении, только в вертикальном или в обоих направлениях

AllowZoom

Позволяет пользователю изменять во время выполнения масштаб изображения, вырезая фрагменты диаграммы или графика курсором мыши (на рис. 4.8 б внизу показан момент просмотра фрагмента графика, целиком представленного на рис. 4.8 а)

Title

Определяет заголовок диаграммы

Foot

Определяет подпись под диаграммой. По умолчанию отсутствует. Текст подписи определяется подсвойством **Text**

Frame

Определяет рамку вокруг диаграммы

Legend

Легенда диаграммы — список обозначений

MarginLeft, MarginRight, MarginTop, MarginBottom

Значения левого, правого, верхнего и нижнего полей

BottomAxis,LeftAxis,RightAxis	Эти свойства определяют характеристики соответственно нижней, левой и правой осей. Задание этих свойств имеет смысл для графиков и некоторых типов диаграмм
LeftWall,BottomWall,BackWall	Эти свойства определяют характеристики соответственно левой, нижней и задней граней области трехмерного отображения графика (см. рис. 4.8 а, нижний график)
SeriesList	Список серий данных, отображаемых в компоненте
View3d	Разрешает или запрещает трехмерное отображение диаграммы
View3DOptions	Характеристики трехмерного отображения
Chart3DPercent	Масштаб трехмерности (для рис. 4.8 это толщина диаграммы и ширина лент графика)

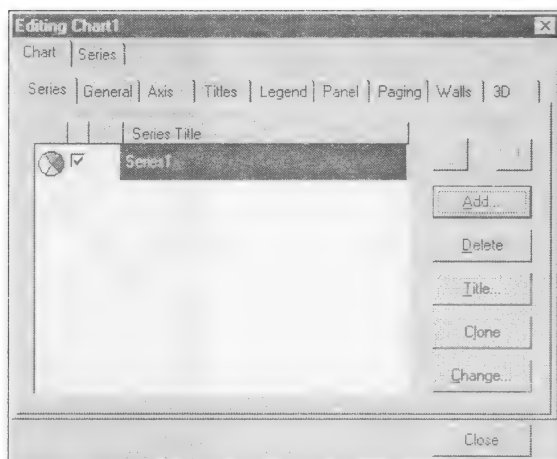
Рядом со многими из перечисленных свойств в Инспекторе Объектов расположены кнопки с многоточием, которые позволяют вызвать ту или иную страницу Редактора Диаграмм — многостраничного окна, позволяющего установить все свойства диаграмм. Вызов Редактора Диаграмм возможен также двойным щелчком на компоненте **Chart** или щелчком на нем правой кнопкой мыши и выбором команды *Edit Chart* во всплывшем меню.

Если вы хотите попробовать воспроизвести приложение, показанное на рис. 4.8, сделайте двойной щелчок на верхнем компоненте **Chart**. Вы попадете в окно Редактора Диаграмм (рис. 4.9) на страницу *Chart*, которая имеет несколько закладок. Прежде всего вас будет интересовать на ней закладка *Series*. Щелкните на кнопке *Add* — добавить серию. Вы попадете в окно (рис. 4.10), в котором вы можете выбрать тип диаграммы или графика. В данном случае выберите *Pie* — круговую диаграмму. Воспользовавшись закладкой *Titles* вы можете задать заголовок диаграммы, закладка *Legend* позволяет задать параметры отображения легенды диаграммы (списка обозначений) или вообще убрать ее с экрана, закладка *Panel* определяет вид панели, на которой отображается диаграмма, закладка *3D* дает вам возможность изменить внешний вид вашей диаграммы: наклон, сдвиг, толщину и т.д.

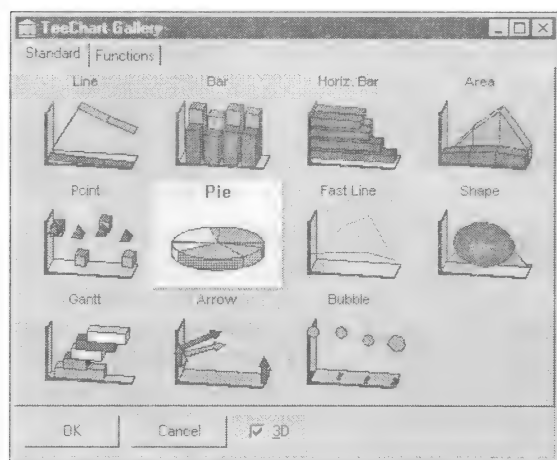
Когда вы работаете с Редактором Диаграмм и выбрали тип диаграммы, в компонентах **Chart** на вашей форме отображается ее вид с занесенными в нее условными данными (см. рис. 4.11).

Рис. 4.9

Редактор Диаграмм,
страница Chart, закладка
Series

**Рис. 4.10**

Выбор типа диаграммы в
Редакторе Диаграмм

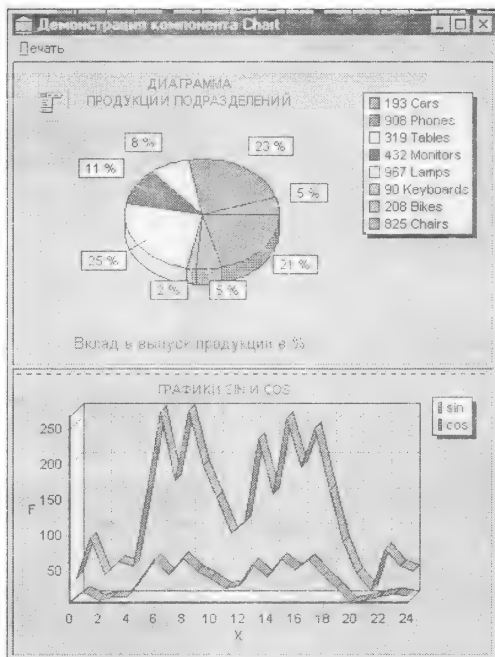


Поэтому вы сразу можете наблюдать результат применения различных опций к вашему приложению, что очень удобно.

Страница Series, также имеющая ряд закладок, дает вам возможность выбрать дополнительные характеристики отображения серии. В частности, для круговой диаграммы на закладке Format полезно включить опцию Circled Pie, которая обеспечит при любом размере компонента Chart отображение диаграммы в виде круга. На закладке Marks кнопки группы Style определяют, что будет написано на ярлычках, относящихся к отдельным сегментам диаграммы: Value — значение, Percent — проценты, Label — названия данных и т.д. В примере рис. 4.8 включена кнопка Per-

Рис. 4.11

Форма приложения рис. 4.8 с занесенными в нее условными данными



cent, а на закладке General установлен шаблон процентов, обеспечивающий отображение только целых значений.

Вы можете, если хотите, добавить на этот компонент Chart еще одну тождественную серию, нажав на закладке Series страницы Chart кнопку Clone, а затем для этой новой серии нажать кнопку Change (изменить) и выбрать другой тип диаграммы, например, Bar. Конечно, два разных типа диаграммы на одном рисунке будут выглядеть плохо. Но вы можете выключить индикатор этой новой серии на закладке Series, а потом предоставить пользователю выбрать тот или иной вид отображения диаграммы (ниже будет показано, как это делается).

Выйдите из Редактора Диаграмм, выделите в вашем приложении нижний компонент Chart и повторите для него задание свойств с помощью Редактора Диаграмм. В данном случае вам надо будет задать две серии, если хотите отображать на графике две кривые, и выбрать тип диаграммы Line. Поскольку речь идет о графиках, вы можете воспользоваться закладками Axis и Walls для задания координатных характеристик осей и трехмерных граней графика.

На этом проектирование внешнего вида приложения завершается. Осталось написать код, задающий данные, которые вы хотите отображать. Для тестового приложения давайте зададим в круговой диаграмме просто некоторые константные данные, а в графиках — функции синус и косинус.

Для задания отображаемых значений надо использовать методы серий **Series**. Остановимся только на трех основных методах.

Метод **Clear** очищает серию от занесенных ранее данных.

Метод **Add**:

```
Add(Const AValue:Double; Const ALabel:String;  
      AColor:TColor)
```

позволяет добавить в диаграмму новую точку. Параметр **AValue** соответствует добавляемому значению, параметр **ALabel** — название, которое будет отображаться на диаграмме и в легенде, **AColor** — цвет. Параметр **ALabel** — не обязательный, его можно задать пустым: ''.

Метод **AddXY**:

```
AddXY(Const AXValue, AYValue: Double;  
       Const ALabel: String; AColor: TColor)
```

позволяет добавить новую точку в график функции. Параметры **AXValue** и **AYValue** соответствуют аргументу и функции. Параметры **ALabel** и **AColor** те же, что и в методе **Add**.

Таким образом, процедура, обеспечивающая загрузку данных в наш пример, может иметь вид:

```
const A1=155;  
      A2=251;  
      A3=203;  
      A4=404;  
  
var i:word;  
begin  
  With Series1 do  
  begin  
    Clear;  
    Add(A1, 'Цех 1', clYellow);  
    Add(A2, 'Цех 2', clBlue);  
    Add(A3, 'Цех 3', clRed);  
    Add(A4, 'Цех 4', clPurple);  
  end;  
  Series2.Clear;  
  Series3.Clear;  
  for i:=0 to 100 do begin  
    Series2.AddXY(0.02*Pi*i, sin(0.02*Pi*i), '', clRed);  
    Series3.AddXY(0.02*Pi*i, cos(0.02*Pi*i), '', clBlue);  
  end;  
  
end;
```

Эту процедуру можно включить в обработку щелчка какой-нибудь кнопки, в команду меню или просто в событие **OnCreate** формы. Операторы **Clear** нужны, если в процессе работы приложения вы собираетесь обновлять данные. Без этих операторов повторное выполнение методов **Add** и **AddXY** только добавит новые точки, не удалив прежние.

Если вы предусмотрели, например, для данных, отображаемых в диаграмме, две серии **Series1** и **Series4** разных видов — Pie и Bar, то можете ввести процедуру, изменяющую по требованию пользователя тип диаграммы. Эту процедуру можно ввести в событие **OnClick** какой-нибудь кнопки, в команду меню или, например, просто в обработку щелчка на компоненте **Chart**. Для того, чтобы загрузить данные в **Series4** и сделать эту диаграмму в первый момент невидимой, можно вставить в конце приведенной ранее процедуры операторы

```
Series4.Assign(Series1);  
Series4.Active:=false;
```

Первый из этих операторов переписывает данные, помещенные в **Series1**, в серию **Series4**. А второй оператор делает невидимой серию **Series4**. Смена типа диаграммы осуществляет процедура

```
Series1.Active:= not Series1.Active;  
Series4.Active:= not Series4.Active;
```

На рис. 4.8 б вы можете видеть результат переключения пользователя на другой вид диаграммы.

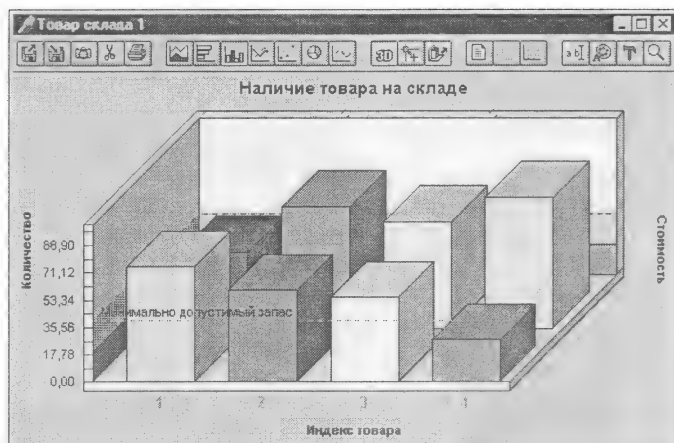
4.6.2 Компоненты страницы **ActiveX** — **F1Book**, **Chartfx**, **Graph**

К компонентам, отображающим графическую информацию, может быть отнесен рассмотренный в разделе 3.4.4 компонент **F1Book** со страницы библиотеки **ActiveX**. На приводившемся в этом разделе рис. 3.14 показано, как можно использовать этот компонент для отображения диаграмм и графиков данных, занесенных в таблицу.

Еще одним интересным компонентом, отображающим графическую информацию, является **Chartfx** со страницы библиотеки **ActiveX**. Он представляет собой законченный редактор диаграмм со встроенной инструментальной панелью (рис. 4.12). Нажимая кнопки инструментальной панели пользователь может задавать новые данные (вторая кнопка справа на рис. 4.12 и выбор опции **Data Editor**), изменять тип диаграммы, сохранять диаграмму в файле с расширением **.chf** или загружать ее из аналогичного файла, копировать диаграмму в буфер обмена **Clipboard** и таким образом включать ее в другие документы (например, в документы **Word**) и т.п.

Рис. 4.12

Приложение
на основе
компонента
Chartfx

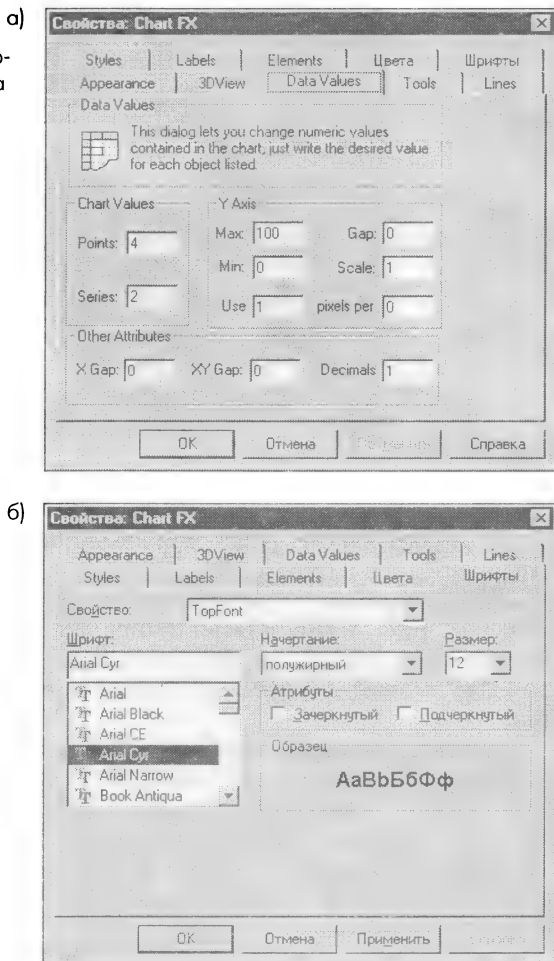


Как и в других компонентах ActiveX, доступ к свойствам **Chartfx** во время проектирования может осуществляться с помощью Инспектора Объектов или щелчком правой кнопки мыши и выбором из всплывшего меню команды **Properties** (Свойства). При выборе этой команды вы попадете в многостраничное диалоговое окно (см. рис. 4.13), позволяющее задать свойства компонента. Остановимся только на нескольких из них, которые вы можете задавать в этом диалоге, в Инспекторе Объектов или программно.

Свойство **Series** на странице **Data Values** (рис. 4.13 а) диалога (в Инспекторе Объектов это свойство названо **NSeries**) обозначает число серий данных (на рис. 4.12 равно 2). Свойство **Points** на той же странице диалога (в Инспекторе Объектов оно названо **NValues**) обозначает число значений по оси аргументов (на рис. 4.12 равно 4). Страница диалога **Elements** позволяет задать какие-то характерные уровни (опции **Value — Text**, на рис. 4.12 отмечен уровень 40 с текстом «Минимально допустимый запас»), выделить цветом какие-то полосы уровней (опции **From — To — Color**, на рис. 4.12 выделены полосы 0 — 20 и 20 — 40), задать текст в строке состояния (опции **ID — Width — Text**). Прочие свойства позволяют задать тексты сверху диаграммы, внизу, слева, справа, задать координатные сетки и многое другое. Следует обратить внимание на выбор шрифтов на странице **Шрифты** (рис. 4.13 б). Шрифты, естественно, надо выбрать такие, которые содержат символы кириллицы. При выборе шрифтов надо сначала в выпадающем списке Свойство выбрать надпись, для которой указывается шрифт (например, **TopFont** — шрифт надписи над диаграммой), а затем в окнах Шрифт, Начертание, Размер установить атри-

Рис. 4.13

Окно задания свойств компонента Chartfx: страницы Data Values (а) и Шрифты (б)



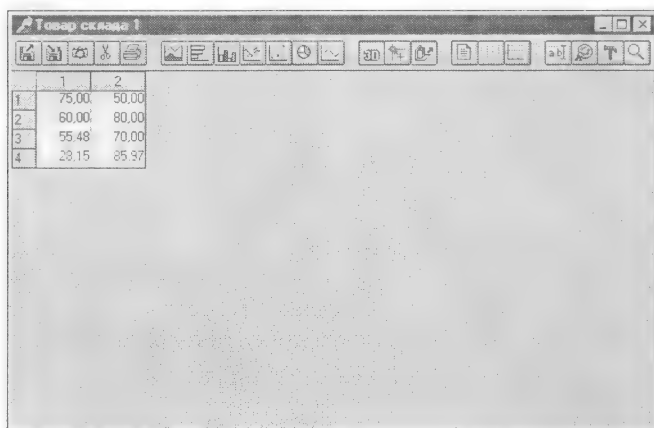
буты шрифта. Подобную процедуру надо повторить для всех надписей.

Теперь давайте коротко рассмотрим некоторые кнопки инструментальной панели компонента во время выполнения приложения (см. рис. 4.12). Первая и вторая слева быстрые кнопки обеспечивают соответственно чтение и сохранение диаграммы. Диаграмма сохраняется в файле с расширением **.chf** и может быть прочитана в последующих сеансах работы. Третья кнопка слева заносит диаграмму в буфер обмена Clipboard, откуда ее можно взять в каком-то другом приложении (например, в Word)

и вставить в документ. Кнопки в центральной части панели позволяют изменять тип диаграммы или графика. Поэкспериментируйте с ними и вы легко поймете, что они означают. Вторая справа группа кнопок позволяет вводить на диаграмме или графике координатную сетку. Правая группа кнопок обеспечивает задание надписей на изображении, выбор шрифта надписей и т.п. Главной из этих кнопок является вторая справа. Она вызывает выпадающее меню, содержащее, в частности, раздел **Data Editor**. Если вы выберете этот раздел, вместо диаграммы вы увидите окно редактора данных, отображаемых на графике или в диаграмме (см. рис. 4.14). Сделав двойной щелчок на том или ином числе, вы можете изменить его. После того, как вы отредактировали данные, опять щелкните на второй справа кнопке инструментальной панели и снимите выделение с раздела **Data Editor**. Вы опять увидите диаграмму, отображающую введенные вами данные.

Рис. 4.14

Редактор отображаемых данных



	1	2
1	75.00	50.00
2	60.00	80.00
3	55.48	70.00
4	28.15	85.97

Более простым, но, возможно, более полезным на странице библиотеки **ActiveX** является компонент **Graph** (рис. 4.15). Он позволяет отображать данные в виде диаграмм различных типов. Настройку можно проводить так же, как и в других компонентах **ActiveX**, или в Инспекторе Объектов, или командой **Properties** из меню, всплывающего при щелчке правой кнопкой мыши на компоненте. В многостраничном диалоговом окне, всплывающем при выполнении этой команды, на странице **Graph** можно выбрать тип диаграммы или графика (опция **GraphType**) и число точек аргумента (**NumPoints**). На странице **Data** можно задать данные: **GraphData** — значения функции в различных точках,

XPosData — значения аргументов в этих точках, **ColorData** — значения цветов в точках (для диаграмм). Остальные опции достаточно похожи на те, которые рассматривались для предыдущих компонентов.

Рис. 4.15

Приложение на основе компонента Graph



Во время выполнения приложения задать новые данные, отредактировать их, настроить график или диаграмму можно методом **BrowseProperties**. Например, оператор

```
Graph1.BrowseProperties;
```

вызовет то же многостраничное окно свойств, которое вы видите во время проектирования, выполняя команду Properties из всплывающего меню компонента. В этом окне можно задать или отредактировать всю отображаемую компонентом информацию. Изменять во время выполнения данные и способ их отображения можно также задавая значения свойствам компонента, большинство из которых вы можете видеть в процессе проектирования в окне Инспектора Объектов.

4.7 Печать

4.7.1 Методы компонентов, обеспечивающие печать

Ряд компонентов, описанных в предыдущих разделах, имеют методы, обеспечивающие печать хранящихся в них данных. Например, компонент **RichEdit** (см. раздел 3.3.4) имеет метод **Print**, позволяющий печатать хранящийся в нем текст. В этот метод передается единственный параметр типа строки, назначение которого заключается только в том, что при просмотре в Windows очереди печатаемых заданий принтера эта строка появляется как имя задания. Например, оператор

```
RichEdit1.Print('Печать RichEdit1');
```

обеспечивает печать текста компонента **RichEdit1**, причем задание на печать получает имя «Печать RichEdit1».

Печать воспроизводит все заданные особенности форматирования. Перенос строк и разбиение текста на страницы производится автоматически. Длина строк никак не связана с размерами компонента **RichEdit**, содержащего этот текст.

Компонент **Chart** (см. раздел 4.6.1) также имеет метод **Print**, обеспечивающий печать отображаемого в компоненте графика или диаграммы. Предварительно может быть выполнен метод **PrintPortrait**, задающий книжную (вертикальную) ориентацию бумаги, или метод **PrintLandscape**, задающий альбомную (горизонтальную) ориентацию. Масштабировать размер печатаемого графика можно, вызвав предварительно метод **PrintRect**:

```
procedure PrintRect ( Const R : TRect ) ;
```

в котором параметр **R** определяет размер области принтера, в которой осуществляется печать.

Компонент **Chartfx** (см. раздел 4.6.2) имеет быструю кнопку печати (пятая слева в инструментальной панели рис. 4.12), с помощью которой пользователь в любой момент может напечатать текущий график или диаграмму.

Не только многие компоненты, но и сами формы имеют метод **Print**, который печатает клиентскую область формы. При этом полоса заголовка формы и полоса главного меню не печатается. Таким образом, можно включить в приложение форму, в которой пользователь во время выполнения размещает необходимые для печати результаты. Если имя этой формы **Form2**, то ее печать может выполняться оператором

```
Form2.Print;
```

Свойство формы **PrintScale** определяет опции масштабирования изображения при печати. Возможные значения **PrintScale**:

poNone	Масштабирование не используется. Размер изображения может изменяться в зависимости от используемого принтера
poPrintToFit	Делается попытка напечатать изображение формы того же размера, который виден на экране
poProportional	Увеличивает или уменьшает размер изображения, подгоняя его под размер страницы, заданный при установке принтера. Это значение принято по умолчанию

4.7.2 Печать с помощью объекта **Printer**

В Delphi имеется класс печатающих объектов **TPrinter**, который обеспечивает печать текстов, изображений и других объектов, расположенных на его канве — **Canvas** (см. разделы 3.2 и 4.2). На канве объекта типа **TPrinter** могут размещаться и тексты, и изображения.

Модуль Delphi, именуемый **Printers**, содержит переменную **Printer**, являющуюся объектом типа **TPrinter**. Эта переменная эквивалентна невизуальному компоненту, только отсутствующему в палитре библиотеки. Чтобы использовать **Printer**, надо добавить модуль **Printers** в оператор **uses** вашей программы. Автоматически он не добавляется.

Рассмотрим некоторые свойства и методы объекта типа **TPrinter**.

Свойство, метод	Описание
Canvas	Канва Canvas — место в памяти, в котором формируется страница или документ перед печатью. Canvas обладает рядом свойств, включая Pen (перо) и Brush (кисть), которые позволяют вам делать рисунки и помещать на них текст. Подробное описание канвы и методов работы с ней вы найдете в разделе 4.2
TextOut	Метод канвы, который позволяет посылать в нее текст (см. раздел 3.2)
BeginDoc	Используется для начала задания печати
EndDoc	Используется для окончания задания печати. Фактическая печать происходит только при вызове EndDoc
PageHeight	Возвращает высоту страницы в пикселях
NewPage	Принудительно начинает новую страницу на принтере
PageNumber	Возвращает текущий номер печатаемой страницы

Предположим, вы хотите напечатать текст, используя печатающий объект. Вы можете написать код вида:

```
Printer.BeginDoc;
Printer.Canvas.TextOut(10,10,
                        'Я печатаю через объект Printer');
Printer.EndDoc;
```

Этот код вызывает печать на канве принтера текста «Я печатаю через объект Printer», начиная с десятого пикселя слева и де-

сятого сверху. **BeginDoc** запускает задание на печать. Текст посылается на канву с помощью метода **TextOut** объекта **Canvas**. Метод **EndDoc** вызывает печать текста и останавливает задание на печать.

Если вы хотите напечатать изображение, хранящееся в компоненте **Image1** (см. раздел 4.3), это можно сделать операторами:

```
Printer.BeginDoc;
with Image1.Picture.Bitmap do
  Printer.Canvas.CopyRect(Rect(0,0,Height,Width),
                        Canvas,Rect(0,0,Height,Width));
Printer.EndDoc;
```

При печати текста объект **Printer** не производит автоматического переноса строк и разбиения текста на страницы. Поэтому печать длинных текстов с помощью объекта **Printer** требует достаточно сложного программирования. Проще это делать, например, загрузкой текста в компонент **RichEdit** (см. раздел 3.3.4) и выполнением его метода **Print** (см. раздел 4.7.1). Другой возможный вариант — использование описанной в следующем разделе системы **QuickReport**.

4.7.3 Подготовка и печать отчетов с помощью **QuickReport** — компоненты **QuickRep**, **QRSubDetail**, **QRLabel**, **QRExpr**, **QRSysData**, **QRMemo**, **QRRichText**, **QRShape**, **QRImage**

На странице **QReport** палитры компонентов **Delphi** имеется множество элементов (см. таблицу 4.2), предназначенных для подготовки и печати отчетов с помощью системы **QuickReport**.

Таблица 4.2. Компоненты системы **QuickReport**

Пикто- грамма	Компонент	Описание
	QuickRep (отчет)	Используется для введения в приложение средств печати отчетов QuickReport
	QRSubDetail (детали)	Используется для компоновки в отчет дополнительных данных
	QRStringsBand (полоса текста)	Используется для компоновки в отчет дополнительных текстов
	QRBand (полоса)	Используется для построения отчетов путем размещения на нем печатаемых компонентов

Пикто- грамма	Компонент	Описание
	QRChildBand (дочерняя полоса)	Используется для создания дочерних полос, которые могут содержать другие компоненты QuickRep и полосы
	QRGroup (группировка)	Используется для группировки данных
	QRLabel (метка)	Используется для размещения текста в отчете
	QRDBText (текст из базы данных)	Представляет собой ориентированный на данные компонент для размещения текста в отчете
	QRExpr (математические выражения)	Позволяет вам строить и отображать выражения над полями данных и системными величинами (такими, как время и дата). Свойство expression компонента включает диалоговое окно Построителя Выражений (Expression Builder), дающее возможность графического построения выражений
	QRSysData (системные данные)	Используется для отображения системных данных
	QRMemo (многострочный текст)	Используется для размещения в отчете многострочных текстов
	QRExprMemo (тексты с математическими выражениями)	Используется для размещения в отчете текстов с математическими выражениями
	QRRichText (многострочный текст RTF)	Используется для размещения в отчете текста в обогащенном формате
	QRDBRichText (многострочный текст RTF базы данных)	Используется для размещения в отчете текста из базы данных в обогащенном формате
	QRShape (форма)	Используется для рисования в отчете графических форм
	QRImage (изображение)	Используется для печати изображений в отчете
	QRDBImage (изображение из базы данных)	Используется для печати изображений из баз данных в отчете

Пиктограмма	Компонент	Описание
	QRCompositeReport (составной отчет)	Используется для построения составных отчетов
	QRPreview (предварительный просмотр)	Используется для предварительного просмотра на экране подготовленного к печати отчета
	QRTextFilter (фильтр текста)	Используется для установки фильтра текста
	QRCSVFilter (разделитель)	Используется для установки разделителя текста
	QRHTMLFilter (фильтр HTML)	Используется для установки фильтра HTML
	QRChart (диаграммы, графики)	Используется для печати в отчете диаграмм, построенных на основе баз данных

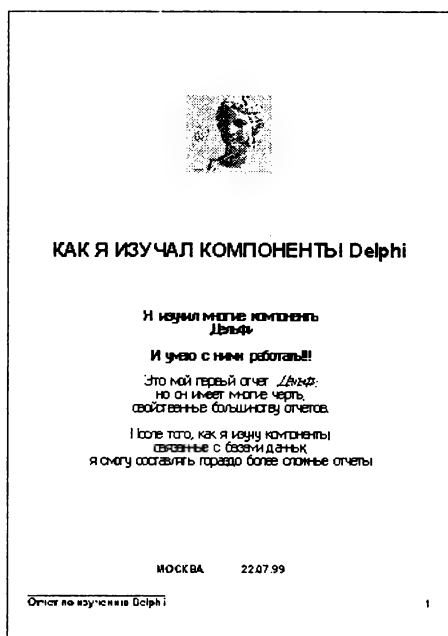
В данном разделе мы коротко рассмотрим принципы создания отчетов с помощью QuickReport. Конечно, рассмотреть в деталях все компоненты QuickReport невозможно — это потребовало бы отвести под эту тему целую главу, а, возможно, и несколько. К тому же отчеты обычно строятся для того, чтобы распечатывать информацию из баз данных. А рассмотрение компонентов, связанных с данными, выходит за рамки данной книги. Поэтому ограничимся только самыми общими принципами построения отчетов, не связанных с базами данных.

QuickReport — это система, позволяющая визуально проектировать отчеты и связывать их с кодом приложения. QuickReport генерирует отчеты, представляемые в виде полос различных типов. Именно для размещения в отчете полос предназначены такие компоненты, как **QRSubDetail**, **QRStringsBand**, **QRBand**, **QRChildBand**, **QRGroup**. Однако, понятие полосы тесно связано с отображением информации из баз данных. С этим же связаны такие компоненты, как **QRDBText**, **QRDBRichText**, **QRDBImage**. Все они в совокупности позволяют строить отчеты, в которых автоматически просматриваются базы данных и отображается их текущее состояние. Но в данном разделе мы сосредоточимся на основном компоненте **QuickRep** и на компонентах, не связанных с данными. При этом можно строить только простые отчеты, содержащие тексты и рисунки, как показано на рис. 4.16.

Основным компонентом, на котором строится весь отчет, является **QuickRep**. Он предоставляет ряд возможностей по управ-

Рис. 4.16

Пример отчета



лению создаваемым отчетом, включая формирование заголовка, полос, шрифтов, установок принтера и др. Этот компонент является визуальным и после его соединения с базой данных может использоваться как контейнер полос, составляющих отчет.

Компонент **QuickRep** имеет ряд свойств, определяющих характеристики печати отчета:

PrinterSetting	Задаёт число копий отчета и диапазон печатаемых страниц
Page	Задаёт размер страницы PaperSize (можно установить заказной размер — Custom и определить длину и ширину страницы свойствами Length и Width), ее ориентацию и поля
Options	Определяет, надо ли печатать верхний колонтитул первой страницы (FirstPageHeader) и нижний колонтитул последней (LastPageFooter)
Units	Задаёт единицу измерения размеров страницы, полей и т.п.: миллиметры, дюймы, пиксели и т.д.
Zoom	Масштаб печати в процентах
ReportTitle	Заголовок окна предварительного просмотра

Перенесите на форму компонент **QuickRep** и посмотрите перечисленные и иные его свойства в Инспекторе Объектов. Одно из основных свойств — **Bands**. Оно имеет ряд подсвойств. Отметим из них следующие: **HasTitle** — имеется полоса заголовка отчета, **HasDetail** — имеется полоса детализации, **HasPageHeader** — имеется верхний колонтитул (заголовок) на каждой странице отчета, **HasPageFooter** — имеется нижний колонтитул на каждой странице отчета. Для дальнейшего рассмотрения компонентов установите в **true** свойство **HasDetail**. На компоненте **QuickRep** появится слабо видимая полоса с надписью «Detail». На этой полосе, увеличив при необходимости ее размеры, можно размещать компоненты, которые отображают ту или иную информацию: **QRLabel**, **QRExpr**, **QRSysData**, **QRMemo**, **QRRichText**, **QRShape**, **QRImage**.

Компонент **QuickRep** имеет два основных метода: **Preview** — предварительный просмотр, и **Print** — печать. Предварительный просмотр и даже печать отчета можно осуществлять и в процессе проектирования. Для этого надо щелкнуть правой кнопкой мыши на компоненте **QuickRep** и из всплывшего меню выбрать команду **Preview**. Перед вами откроется окно предварительного просмотра, в котором, в частности, имеется кнопка печати.

Компоненты **QRLabel**, **QRMemo**, **QRRichText**, **QRShape**, **QRImage**, размещаемые на полосах отчета, являются аналогами обычных компонентов — **Label**, **Memo**, **RichEdit**, **Shape**, **Image**, которые уже рассматривались ранее. Основной особенностью соответствующих компонентов **QuickReport** является их способность печататься в тех полосах отчета, в которых они размещены. Компоненты имеют два свойства, отсутствующих в обычных компонентах: **Frame** и **Size**.

Свойство **Frame** имеет ряд подсвойств, определяющих рамку вокруг компонента: **Color** — цвет, **Style** — стиль, **Width** — ширина, **DrawBottom**, **DrawLeft**, **DrawRight**, **DrawTop** — определяют наличие рамки соответственно внизу, слева, справа и сверху компонента.

Свойство **Size** имеет подсвойства, определяющие размер и место размещения компонента при печати. Все определяется в единицах измерения, заданных свойством **Units** компонента **QuickRep**.

Некоторые компоненты имеют свойство **AlignToBand** — выравнивание в полосе. Если это свойство установить в **true**, то компонент будет выровнен по краю полосы, заданному свойством **Alignment**: **taLeftJustify** — влево, **taCenter** — по центру, **taRightJustify** — вправо.

Остановимся теперь на свойствах отдельных компонентов. Интересное свойство имеет компонент **QRRichText** — **ParentRichEdit**. Это свойство указывает на обычный компонент **RichEdit**, текст из которого автоматически передается в компонент **QRRichText**. Вы можете, например, ввести в свое приложение компонент **RichEdit** и предоставить пользователю возможность вводить в него нужный текст, форматируя соответствующим образом слова и абзацы. Этот текст будет автоматически переноситься в отчет — в размещенный в нем компонент **QRRichText**. Впрочем, и в другие компоненты **QuickReport** можно программно заносить тексты и изображения, задаваемые пользователем. Например, вы можете ввести в свое приложение компонент **Edit** и предоставить пользователю возможность вводить в него нужный текст. А перед предварительным просмотром или печатью отчета вы можете перенести этот текст, например, в метку **QRLabel**, оператором

```
QRLabel1.Caption:=Edit1.Text;
```

Компонент **QRExpr** позволяет отображать в отчете результаты каких-то вычислений. Основное свойство этого компонента — **Expression**. Нажав на кнопку с многоточием около этого свойства в Инспекторе Объектов, вы попадете в окно, представленное на рис. 4.17 а. В этом окне вы можете ввести выражение, результаты которого должны отображаться в отчете на месте расположения компонента **QRExpr**. Элементами выражения могут быть поля базы данных (Database field), такие переменные (Variable), как текущее время, дата, номер страницы и др., функции (Function). Кнопка **Validate** позволяет вам проверить синтаксическую правильность введенного выражения и посмотреть результат вычисления. Поэкспериментируйте с этим компонентом и вы легко поймете принципы работы с ним.

Пусть, например, вы хотите отобразить текущую дату, включая месяц и год, но не включая числа. Обычно дата отображается в формате «дд.мм.гг». Значит вам надо убрать из этого формата первые три символа, отображающие год. Это можно сделать с помощью функции копирования фрагмента строки **COPY(DATE, 4, 5)**, в которой первый аргумент указывает строку, второй — индекс первого копируемого символа, третий — число копируемых символов. В приведенном примере **DATE** — функция, возвращающая текущую дату в виде строки.

Вы можете просто написать приведенную выше функцию **COPY** с соответствующими аргументами в свойстве **Expression**. Но если вы хотите воспользоваться окном ввода выражений, то вам надо выполнить следующее. Нажмите кнопку **Function**. Перед

Рис. 4.17

Окна ввода выражений компонента QRExp: основное окно (а), окно выбора функции (б) и окно ввода параметров (в)

Expression Wizard

Enter expression:

COPY{DATE, 4, 5}

Insert at cursor position:

Database field	Function	Variable
+	-	*
/	=	<
>	<=	>=
Not	And	Or

Clear Validate OK Cancel

Expression Wizard

Select function

Category	Available functions
All	IF
Date & time	STR
Math & trig	UPPER
Statistical	LOWER
Database	PRETTY
Logical	TIME
Information	DATE
Other	COPY

COPY{<X>, <St>, <Len>}
Returns a substring of string X

Continue Cancel

Expression Wizard

Enter parameters for function:

COPY{<X>, <St>, <Len>}
Returns a substring of string X

Argument 1 - text Value
DATE ...

Argument 2 - numeric value
4 ...

Argument 3 - numeric value
5 ...

OK Cancel

вами откроется окно выбора функции, представленное на рис. 4.17 б. В левом списке вы видите категории функций, а в правом — доступные функции. В нашем примере вам надо выбрать в правом списке функцию **COPY**. При этом под списками вы видите пояснение функции и ее синтаксис. Нажмите кнопку Continue (продолжить). Вы попадете в окно ввода параметров, представленное на рис. 4.17 в. Может в его окнах непосредственно писать значения аргументов. Но если в качестве аргумента выступает другая функция (в нашем примере первый аргумент — функция **DATE**), то можете нажать кнопку с многоточием около окна соответствующего аргумента. Вы опять попадете в окно 4.17 а, можете в нем опять нажать Function и выбрать в появившемся окне рис. 4.17 б функцию **DATE**. После этого вы вернетесь в окно рис. 4.17 в, но в качестве аргумента в нем уже будет занесена функция. Таким образом просто вводить выражения, содержащие вложенные функции.

Когда вы ввели все аргументы, щелкните на ОК и вернетесь в основное окно рис. 4.17 а, в котором будет сформировано требуемое выражение. Щелкните в нем на ОК и сформированное выражение занесется в свойство **Expression**.

Компонент **QRSysData** позволяет отображать в отчете системные данные. Его основное свойство **Data**, которое может принимать следующие значения:

qrsDate	текущая дата
qrsDateTime	текущие дата и время
qrsDetailCount	число записей в базе данных
qrsDetailNo	текущий номер записи в базе данных
qrsPageNumber	номер текущей страницы
qrsReportTitle	заголовок отчета
qrsTime	текущее время

Свойство **Text** компонента **QRSysData** определяет текст, предшествующий отображаемой величине.

Давайте в качестве примера использования всех этих компонентов рассмотрим создание простого отчета вида, представленного на рис. 4.16. Он содержит некоторый текст и рисунки, имеет нижние колонтитулы, в которых печатается название отчета и номер страницы. Кроме того, мы хотим, чтобы на первой страни-

це над нижним колонтитулом было написано «МОСКВА» и дата создания отчета.

- Начните новый проект.
- Перенесите на форму компонент **QuickRep** и в его свойстве **Bands** установите в **true** подсвойство **HasDetail** — основная полоса, в которой вы будете писать текст, и подсвойство **HasPageFooter** — полоса нижних колонтитулов.

Свойство **HasTitle** можно устанавливать в **true**, а можно не устанавливать, поскольку различие между заголовком отчета и его полосой детализации проявляться в нашем случае не будет. Дело в том, что при работе с базами данных полоса детализации может печататься многократно и тогда надо отдельно выделить полосу заголовка, которая должна печататься только один раз. А в нашем случае полоса детализации будет печататься один раз, так что она не отличается от заголовка.

Если вы установили в **true** свойства **HasDetail** и **HasPageFooter**, то на поле компонента **QuickRep** появятся слабо видимые полосы с надписями «Detail» и «Page Footer».

- Перенесите в верхнюю часть полосы детализации компонент **QRImage**. Щелкните на кнопке с многоточием около его свойства **Picture** и загрузите в него какую-нибудь картинку так же, как в обычный компонент **Image** (см. раздел 4.3).
- Перенесите в полосу детализации компонент **QRLabel**, расположите его под рисунком и задайте заголовок отчета в свойстве **Caption**. Установите свойство **Alignment** в **taCenter** и **AlignToBand** в **true**. Это обеспечит размещение заголовка в центре страницы по горизонтали. Назовите эту метку **QRTitle**. В дальнейшем нам надо будет на нее ссылаться.
- Разместите примерно в центре страницы компонент **QRRichText**, в котором будет писаться основной текст. Напишите какой-нибудь текст в свойстве **Lines**.

Теперь на страницу надо добавить строку с текстом «МОСКВА» и датой создания отчета. Но если мы разместим ее под компонентом **QRRichText**, то в отчете она окажется непосредственно под текстом. А мы хотим разместить ее внизу страницы. Поэтому нам надо ввести в отчет еще одну полосу.

- Перенесите на форму компонент **QRSubDetail**. Установите его свойство **AlignToBottom** в **true**. Это обеспечит размещение полосы внизу страницы.
- Разместите на полосе **Detail** компонент **QRLabel**, написав в его свойстве **Caption** текст «МОСКВА».

- Разместите правее метки компонент **QRSysData**. В его свойстве **Data** установите значение **qrsDate**. Этот компонент будет отображать дату создания отчета.

Это не лучший вариант, так как при этом будет отображаться день, месяц и год. Если хотите, чтобы отображались только месяц и год, используйте вместо **QRSysData** компонент **QRExpr**, в котором можно, как показано выше, задать отображение даты без указания дня.

Осталось внести информацию в нижний колонтитул страницы.

- Перенесите на полосу «Page Footer» компонент **QRLabel**, задайте в его свойстве **Caption** текст (например, «Отчет по изучению Delphi»). Установите свойство **Alignment** в **taLeftJustify** и **AlignToBand** в **true**. Это обеспечит выравнивание метки по левому краю страницы. Установите в свойстве **Frame** подсвойство **DrawTop** в **true**. Это обеспечит появление линии, отделяющей колонтитул от текста.

- Перенесите на полосу «Page Footer» компонент **QRSysData**. В его свойстве **Data** установите значение **qrsPageNumber**. Этот компонент будет отображать номер страницы. Установите свойство **Alignment** в **taRightJustify** и **AlignToBand** в **true**. Это обеспечит выравнивание номера по правому краю страницы.

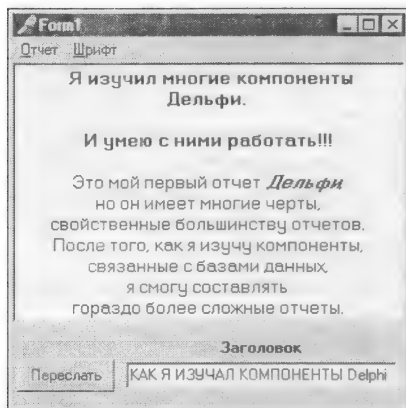
Проектирование отчета завершено. Щелкните правой кнопкой мыши на компоненте **QuickRep** и из всплывшего меню выберите команду **Preview**. Перед вами откроется окно предварительного просмотра, в котором вы увидите свой отчет. Подберите размещение компонентов и размеры шрифтов в них так, чтобы все выглядело прилично.

Конечно, подобный отчет совсем не интересен, так как он весь создан в процессе проектирования. К тому же вы никак не использовали возможностей форматирования текста в компоненте **QRRichText**. Поэтому текст выглядит некрасиво. Давайте теперь создадим форму (рис. 4.18), которая управляла бы построением, просмотром и печатью этого отчета.

- Дайте имя **FRep** вашей форме (свойство **Name**). По этому имени вы потом сможете ссылаться на ее компоненты.
- Проверьте, что свойство **Visible** этой формы установлено в **false**. Это необходимо сделать, так как форма в дальнейшем должна быть невидимой.

Рис. 4.18

Окно формы, управляющей подготовкой отчета



- Сохраните проект, дав файлу формы имя **URep**. Это имя автоматически присвоится модулю текста формы и по нему вы сможете впоследствии сослаться на этот модуль.
- Откройте командой **File | New Form** новую форму.
- Выполните команду **Project | Options**, в открывшемся окне Опций проекта перейдите на страницу **Forms** и в списке **Main Form** выберите главной текущую форму **Form1**. Форма **FRep** должна быть вспомогательной.
- Сохраните модуль новой формы, дав его файлу имя **URep1**. По этому имени вы сможете сослаться на него из модуля **URep**.
- Перенесите на новую форму компоненты главное меню **Main-Menu**, **FontDialog**, **RichEdit**, **Edit** и **Button**. Расположите их примерно так, как показано на рис. 4.18. В меню введите раздел **Отчет** с подразделами **Показать** и **Печать**, и раздел **Шрифт**. В свойство **Lines** компонента **RichEdit** введите некоторый начальный текст, например, тот, который вы ранее вводили в **QRRichText**. В окно редактирования **Edit** введите текст заголовка вашего отчета.

Теперь осталось связать друг с другом две формы и написать небольшие команды управления.

- Перейдите в модуль формы отчета **URep** и в его разделе **implementation** напишите оператор

```
uses URep1;
```

Этот оператор позволит сослаться из модуля отчета на компоненты главной формы.

- Перейдите в модуль главной формы **URep1** и в его разделе **implementation** напишите оператор

uses URep;

Этот оператор позволит ссылаться из модуля главной формы на компоненты модуля отчета.

- В компоненте **QRRichText** формы **URep** раскройте выпадающий список в свойстве **ParentRichEdit**. В этом списке должна появиться ссылка на компонент **RichEdit** формы **Form1**. Установите это свойство, чтобы связать окна редактирования друг с другом.

- В обработчик события **OnClick** кнопки на форме **Form1** внесите оператор

```
FRep.QRLTitle.Caption:=Edit1.Text;
```

Он обеспечит пересылку в метку заголовка отчета **QRLTitle** текста, который пользователь ввел в окно **Edit1**.

- В обработку команды меню **Шрифт** вставьте операторы

```
if FontDialog1.Execute then  
  RichEdit1.SelectAttributes.Assign(FontDialog1.Font);  
RichEdit1.SetFocus;
```

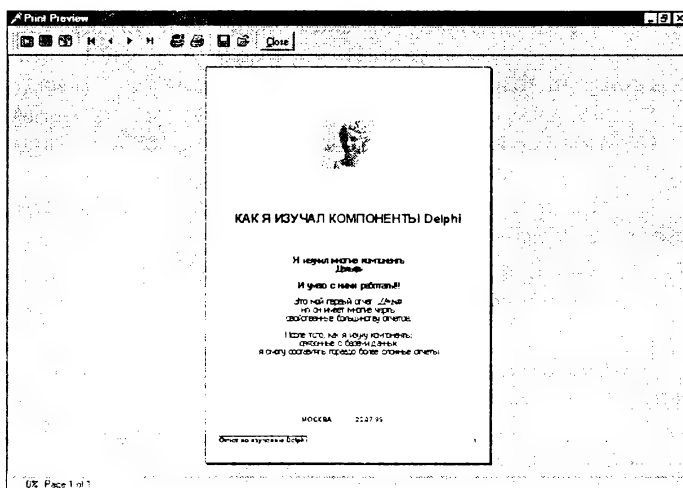
Эти операторы, обеспечивающие форматирование в окне **RichEdit1**, уже рассматривались в разделе 3.3.4.

- В обработку команды меню **Просмотр** вставьте оператор

```
FRep.QuickRep1.Preview;
```

Приложение закончено. Запустите его на выполнение. Отформатируйте текст в окне (см. пример на рис. 4.18). Выполните команду **Просмотр**. Вы увидите окно предварительного просмотра

Рис. 4.19
Предварительный просмотр отчета



ра, показанное на рис. 4.19. В этом окне вы можете просматривать отчет, изменяя размер изображения, переходить со страницы на страницу (ваш отчет имеет всего одну страницу, так что переходы в нем невозможны), сохранять отчет в файле, проводить установку принтера, печатать отчет, сохранять его в файле, чтобы в каких-то будущих сеансах можно было загрузить его опять из файла. Словом, возможностей много. Опробуйте их все.

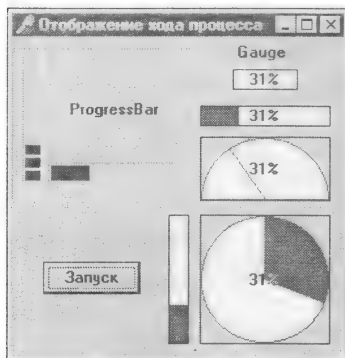
4.8 Мультимедиа и анимация

4.8.1 Отображение хода выполнения длительных операций — компоненты **ProgressBar** и **Gauge**

Рассмотрим компоненты **ProgressBar** со страницы библиотеки Win32 и **Gauge** со страницы Samples, предназначенные для отображения в стиле Windows 95/98 хода процессов, занимающих заметное время, например, копирования больших файлов, настройку приложения, установку приложения на компьютере и т.п. Пример возможных вариантов отображения хода процесса компонентами **ProgressBar** и **Gauge** приведен на рис. 4.20.

Рис. 4.20

Пример отображения хода процесса компонентами **ProgressBar** и **Gauge**



Основные свойства этих компонентов очень схожи, различаясь только именами:

Свойство ProgressBar	Свойство Gauge	Описание
Max	MaxValue	Максимальное значение позиции (Position , Progress), которое соответствует завершению отображаемого процесса. По умолчанию задается в процентах — 100
Min	MinValue	Начальное значение позиции (Position , Progress), которое соответствует началу отображаемого процесса
Position	Progress	Позиция, которую можно задавать по мере протекания процесса, начиная со значения Min или MinValue в начале процесса, и кончая значением Max или MaxValue в конце. Если минимальное и максимальное значения выражены в процентах, то позиция — это процент завершенной части процесса
Smooth	—	Непрерывное (при значении true) или дискретное отображение процесса. На рис. 4.20 в горизонтальном компоненте ProgressBar задано Smooth = true , а в вертикальном — false
Step	—	Шаг приращения позиции, используемый в методе StepIt . Значение по умолчанию — 10
Orientation	—	Ориентация шкалы компонента: pbHorizontal — горизонтальная, pbVertical — вертикальная. Если задана ориентация pbVertical , то компонент надо вытянуть по вертикали (см. на рис. 4.20 компонент слева)
—	ForeColor	Цвет заполнения
—	ShowText	Текстовое отображение процента выполнения на фоне диаграммы
—	Kind	Тип диаграммы: gkHorizontalBar — горизонтальная полоса, gkVerticalBar — вертикальная полоса, gkPie — круговая диаграмма, gkNeedle — секторная диаграмма, gkText — отображение текстом

Отображение хода процесса можно осуществлять, задавая значение позиции — **Position** в **ProgressBar** или **Progress** в **Gauge**. Например, если полная длительность процесса характеризуется значением целой переменной **Count** (объем всех копируемых файлов, число настроек, количество циклов какого-то процесса), а выполненная часть — целой переменной **Current**, то задавать позицию диаграммы в случае, если используются значения ми-

нимальной и максимальной позиции по умолчанию (т.е. 0 и 100), можно операторами

```
ProgressBar1.Position:=100 * Current div Count;
```

или

```
Gauge1.Progress:=100 * Current div Count;
```

соответственно для **ProgressBar** и **Gauge**.

Можно поступать иначе: задать сначала значение максимальной величины равным **Count**, а затем в ходе процесса задавать позицию равной **Current**. Например:

```
Gauge1.MaxValue:=Count;
```

```
...
```

```
Gauge1.Progress:= Current;
```

Компонент **ProgressBar** имеет два метода, которыми тоже можно воспользоваться для отображения процесса: **StepBy(Delta: Integer)** — увеличение позиции на заданную величину **Delta**, и **StepIt** — увеличение позиции на один шаг, величина которого задается свойством **Step**.

4.8.2 Воспроизведение немых видео клипов — компонент **Animate**

Теперь рассмотрим способ воспроизведения в приложении Delphi стандартных мультипликаций Windows и файлов **.avi** — клипов без звукового сопровождения. Это позволяет сделать компонент **Animate**, расположенный на странице Win32 библиотеки.

Компонент **Animate** позволяет воспроизводить на форме стандартные видео клипы Windows (типа копирования файлов, поиска файлов и т.п.) и немые видео файлы **.avi** — Audio Video Interleaved. Эти файлы представляют собой последовательность кадров битовых матриц. Они могут содержать и звуковую дорожку, но компонент **Animate** воспроизводит только немые клипы AVI.

Откройте новое приложение, перенесите на форму компонент **Animate** и познакомьтесь с ним.

Воспроизводимое им изображение задается одним из двух свойств: **FileName** или **CommonAVI**. Первое из этих свойств, как ясно из его названия, позволяет в процессе проектирования или программно задать имя воспроизводимого файла. А свойство **CommonAVI** позволяет воспроизводить стандартные мультипликации Windows. Это свойство объявлено следующим образом:

```
type TCommonAVI = (aviNone, aviFindFolder, aviFindFile,
                    aviFindComputer, aviCopyFiles,
                    aviCopyFile, aviRecycleFile,
                    aviEmptyRecycle, aviDeleteFile);
property CommonAVI: TCommonAVI;
```

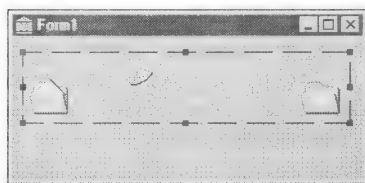
Тип **TCommonAVI** определяет множество предопределенных в Windows мультипликаций типа копирования файлов, поиска файлов, удаления файлов и т.п. Что означает каждое значение вы увидите из тестового приложения, которое постройте чуть позже.

А пока установите значение **CommonAVI**, например, равным **aviCopyFile**. Это соответствует стандартному изображению копирования файла. Соответствующий начальный рисунок немедленно появится на вашей форме. Свойство **Repetitions** компонента **Animate** задает число повторений воспроизведения клипа. Если оно равно 0 (значение по умолчанию), то воспроизведение повторяется вновь и вновь до тех пор, пока не будет выполнен метод **Stop**. При выполнении этого метода генерируется событие **On-Stop**, которое можно использовать, например, чтобы стереть изображение — сделать его невидимым.

Если же свойство **Repetitions** задать большим нуля, оно определит число повторений клипа. Задайте его, например, равным 3. А теперь установите свойство **Active** компонента **Animate** в **true**. Вы увидите (рис. 4.21), что еще в процессе проектирования ваше приложение заработает. Изображение оживет и клип будет повторен 3 раза.

Рис. 4.21

Анимация копирования файла



Вы можете посмотреть воспроизводимое изображение по кадрам. Для этого щелкните на компоненте правой кнопкой мыши и из всплывшего меню выберите разделы **Next Frame** (следующий кадр) или **Previous Frame** (предыдущий кадр). Это позволит вам выбрать фрагмент клипа, если вы не хотите воспроизводить клип полностью. Воспроизвести фрагмент клипа можно, установив соответствующие значения свойств **StartFrame** — начальный кадр воспроизведения, и **StopFrame** — последний кадр воспроизведения.

Воспроизводить фрагмент клипа можно и методом **Play**, который определен следующим образом:

```
procedure Play(FromFrame, ToFrame: Word; Count: Integer);
```

Метод воспроизводит заданную последовательность кадров клипа от **FromFrame** до **ToFrame** включительно и воспроизведе-

ние повторяется **Count** раз. Если **FromFrame** = 1, то воспроизведение начинается с первого кадра. Значение **ToFrame** должно быть не меньше **FromFrame** и не больше значения, определяемого свойством **FrameCount** (свойство только для чтения), указывающим полное число кадров в клипе. Если **Count** = 0, то воспроизведение повторяется до тех пор, пока не будет выполнен метод **Stop**.

Выполнение **Play** идентично заданию **StartFrame** равным **FromFrame**, **StopFrame** равным **ToFrame**, **Repetitions** равным **Count** и последующей установке **Active** в **true**.

В компоненте **Animate** предусмотрены события **OnClose**, **OnOpen**, **OnStart** и **OnStop**, генерируемые соответственно в моменты закрытия и открытия компонента, начала и окончания воспроизведения.

Давайте теперь построим тестовое приложение, показывающее возможности компонента **Animate**. Установите в том приложении, которое вы уже начали, свойство **Visible** компонента **Animate** в **false**. Это надо для того, чтобы изображение возникало только тогда, когда произойдет соответствующее событие: копирование файлов, поиск файлов и т.п. В тестовом приложении мы будем имитировать начало и окончание события, которое должно сопровождаться мультипликацией, нажатиями кнопок запуска и останова воспроизведения. Поэтому верните значение свойства **Repetitions** в 0, чтобы воспроизведение длилось до окончания события. Свойство **Active** установите в **false**. Полезно также установить свойство **AutoSize** в **false**, а свойство **Center** в **true**, чтобы изображение всегда появлялось в центре экрана.

А теперь добавьте в приложение 3 кнопки (рис. 4.22). Первая из них (назовите ее **BWind**) будет начинать процесс воспроизведения поочередно всех стандартных клипов **Windows**. Вторая кнопка (назовите ее **BStop**) пусть завершает воспроизведение очередного клипа. А третью кнопку (назовите ее **BFile**) введем для того, чтобы показать, что компонент может воспроизводить изображения из заданного файла **.avi**. Чтобы пользователь мог выбрать файл изображения, добавьте на форму компонент **OpenDialog** (см. раздел 8.2) и задайте его фильтр (свойство **Filter**) равным

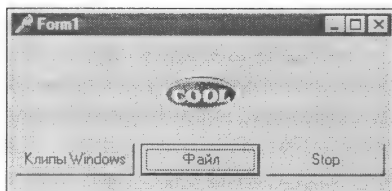
видео *.avi

*.avi

Теперь все приготовления закончены и осталось только написать обработчики событий. Код обработчиков может иметь вид:

Рис. 4.22

Демонстрация возможностей компонента
Animate



```

var i:word;

procedure TForm1.BWindClick(Sender: TObject);
begin
  Animat1.Visible:=true;
  i:=1;
  Animat1.CommonAVI:=aviFindFolder;
  Animat1.Active:=true;
end;

procedure TForm1.BStopClick(Sender: TObject);
begin
  Animat1.Stop;
end;

procedure TForm1.BFileClick(Sender: TObject);
begin
  if OpenFileDialog1.Execute then with Animat1 do begin
    i:=9;
    FileName:=OpenDialog1.FileName;
    Visible:=true;
    Active:=true;
  end;
end;

procedure TForm1.Animat1Stop(Sender: TObject);
begin
  Inc(i);
  with Animat1 do begin
    case i of
      2: CommonAVI:=aviFindFile;
      3: CommonAVI:=aviFindComputer;
      4: CommonAVI:=aviCopyFiles;
      5: CommonAVI:=aviCopyFile;
      6: CommonAVI:=aviRecycleFile;
      7: CommonAVI:=aviEmptyRecycle;
      8: CommonAVI:=aviDeleteFile;
    end;
    if i<<9 then Active:=true
    else Visible:=false;
  end;
end;

```

Обработчик события **OnClick** кнопки **BWind** задает начальное значение свойства **CommonAVI**, сбрасывает счетчик на 1, делает компонент **Animate1** видимым и активизирует его.

Обработчик события **OnClick** кнопки **BStop** останавливает воспроизведение методом **Stop**.

Обработчик события **OnStop** компонента **Animate1** увеличивает счетчик на 1, в зависимости от значения счетчика загружает в компонент соответствующий клип **Windows** и активизирует компонент. Если все клипы уже воспроизведены, то компонент делается невидимым.

Обработчик события **OnClick** кнопки **BFile** загружает в компонент видео файл, выбранный пользователем.

Выполните приложение и проверьте его в работе. В качестве видео файла можете использовать файл ...\\Demos\\Coolstuff\\Cool.avi, поставляемый с примерами Delphi (на рис. 4.22 изображен момент воспроизведения именно этого файла).

4.8.3 Универсальный проигрыватель MediaPlayer

В Delphi (начиная с Delphi 2) имеется компонент **MediaPlayer** — универсальный проигрыватель аудио- и видео-информации. Этот медиа-плеер расположен на странице **System** библиотеки компонентов.

Компонент можно использовать в двух режимах. Во-первых, можно предоставить пользователю возможность управлять воспроизведением информации с помощью кнопочного интерфейса, напоминающего панель управления различными проигрывателями. Во-вторых, можно сделать сам компонент невидимым и управлять воспроизведением информации с помощью его методов.

Пользовательский интерфейс медиа-плеера представлен на рис. 4.23. Он имеет ряд кнопок, управляемых мышью или клавишей пробела и клавишами со стрелками.

Рис. 4.23

Панель компонента MediaPlayer



Назначение кнопок (перечисляются слева направо):

Кнопка	Действие
Play	Воспроизведение
Pause	Пауза воспроизведения или записи. Если медиа-плеер в момент щелчка уже в состоянии паузы, то воспроизведение или запись возобновляются

Кнопка	Действие
Stop	Останов воспроизведения или записи
Next	Переход на следующий трек или на конец
Prev	Переход на предыдущий трек или на начало
Step	Перемещение вперед на заданное число кадров
Back	Перемещение назад на заданное число кадров
Record	Начало записи
Eject	Освобождение объекта, загруженного в устройство

Каждой кнопке медиа-плеера соответствует метод, осуществляющий по умолчанию требуемую операцию: **Play**, **Pause**, **Stop**, **Next**, **Previous**, **Step**, **Back**, **StartRecording**, **Eject**.

Тип устройства мультимедиа, с которым работает медиа-плеер, определяется его свойством **DeviceType**. Если устройство мультимедиа хранит объект воспроизведения в файле, то имя файла задается свойством **FileName**. По умолчанию свойство **DeviceType** имеет значение **dtAutoSelect**. Это означает, что медиа-плеер пытается определить тип устройства исходя из расширения имени файла **FileName**.

Еще одно свойство **MediaPlayer** — **AutoOpen**. Если оно установлено в **true**, то медиа-плеер пытается открыть устройство, указанное свойством **DeviceType**, автоматически во время своего создания в процессе выполнения приложения.

Воспроизведение видео информации по умолчанию производится в окно, которое создает само открытое устройство мультимедиа. Однако это можно изменить, если в свойстве **Display** указать оконный элемент, в котором должно отображаться изображение. Это может быть, например, форма или панель. Можно также задать в свойстве **DisplayRect** типа **TRect** (свойство только времени выполнения) прямоугольную область этого окна, в которую должно выводиться изображение. Для задания свойства **DisplayRect** можно использовать функцию **Rect**. Однако, в данном свойстве использование типа **TRect** не совсем обычно. Первые две координаты, как и обычно, задают положение левого верхнего угла изображения. А два следующих числа задают ширину и высоту изображения, а не координаты правого нижнего угла. Например, оператор

```
MediaPlayer1.DisplayRect:=Rect(10,10,200,200);
```

задает для вывода область с координатами левого верхнего угла (10, 10), длиной и шириной, равными 200.

В компоненте **MediaPlayer** определены события **OnClick** и **OnNotify**. Первое из них происходит при выборе пользователем одной из кнопок медиа-плеера и определено как

```
type
  TMPBtnType=(btPlay, btPause, btStop, btNext, btPrev, btStep,
              btBack, btRecord, btEject);
procedure(Sender:TObject; Button:TMPBtnType;
          var DoDefault: Boolean);
```

Параметр **Button** указывает выбранную кнопку. Параметр **DoDefault**, передаваемый как **var**, определяет выполнение (при значении **true** по умолчанию) или отказ от выполнения стандартного метода, соответствующего выбранной кнопке.

Событие **OnNotify** происходит после возвращения очередного метода, если свойство медиа-плеера **Notify** было установлено в **true**. Способ возврата любого метода медиа-плеера определяется свойством **Wait**. Если установить **Wait** равным **false**, то возвращение управления в приложение происходит сразу после вызова метода, не дожидаясь завершения его выполнения. Таким образом, задав **Notify** равным **true** и **Wait** равным **false**, можно обеспечить немедленный возврат в приложение и отображения пользователю текущего состояния объекта мультимедиа.

Свойства **Notify** и **Wait** действуют только на один очередной метод. Поэтому их значения надо каждый раз восстанавливать в обработчиках событий **OnClick** или **OnNotify**.

В обработчиках событий можно читать свойство **Mode**, характеризующее текущее состояние устройства мультимедиа. Можно также читать и устанавливать ряд свойств, характеризующих размер воспроизводимого файла и текущую позицию в нем.

Вот, собственно, в конспективном виде основная информация о компоненте **MediaPlayer**. А теперь попробуйте все это на практике. Простое и в то же время мощное приложение можно сделать очень легко. Начните новый проект и перенесите на форму компоненты **MediaPlayer**, **MainMenu** и **OpenDialog**. В фильтре компонента **OpenDialog** можно, например, задать:

ауди *.wav, *.mid	*.wav;*.mid
видео *.avi	*.avi
все файлы	.*

В меню достаточно задать одну команду: Файл | Открыть. Обработчик события **OnClick** этой команды может содержать оператор

```

if OpenFileDialog1.Execute
then with MediaPlayer1 do begin
    FileName:=OpenFileDialog1.FileName;
    Open;
end;

```

который открывает устройство мультимедиа, соответствующее выбранному пользователем файлу. При этом надо проследить, чтобы в компоненте **MediaPlayer** свойство **DeviceType** равнялось **dtAutoSelect**. Это обеспечит автоматический выбор соответствующего устройства мультимедиа исходя из расширения выбранного файла.

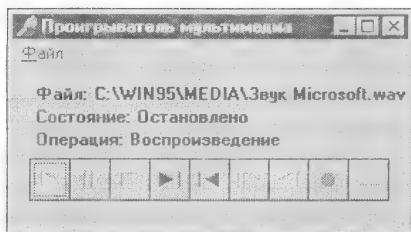
В компоненте **MediaPlayer** при желании можно указать имя файла **FileName**, открываемого в момент начала выполнения приложения. Тогда надо установить свойство **AutoOpen** в **true**. Впрочем, это, конечно, не обязательно.

Вот и все. Можете выполнять свое приложение и наслаждаться музыкой или фильмами (если, конечно, все вопросы, связанные с настройкой мультимедиа на вашем компьютере решены).

Чтобы все-таки использовать какие-то события компонента **MediaPlayer**, давайте немного усложним приложение. Введем в него три метки (рис. 4.24), в которых будем отображать имя воспроизводимого файла, состояние открытого устройства мультимедиа и последнюю вызванную операцию.

Рис. 4.24

Приложение универсального проигрывателя при воспроизведении звукового файла



Код, обеспечивающий подобную обратную связь в приложении, может быть следующим.

```

const ModeStr: array[TMPModes] of string =
    ('Не готово', 'Остановлено', 'Воспроизведение',
     'Запись', 'Поиск', 'Пауза', 'Открыто');
ButtonStr: array[TMPBtnType] of string =
    ('Воспроизведение', 'Пауза', 'Стоп',
     'Следующий', 'Предыдущий', 'Вперед', 'Назад',
     'Запись', 'Конец');

procedure TForm1.OpenClick(Sender: TObject);
begin
    if OpenFileDialog1.Execute

```

```

then with MediaPlayer1 do begin
  FileName:=OpenDialog1.FileName;
  Label1.Caption:='Файл: '+FileName;
  Open;
  Notify:=true;
end;
end;

procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  with MediaPlayer1 do begin
    Label2.Caption := 'Состояние: '+ModeStr[Mode];
    {Переустановка Notify, чтобы событие произошло
    в следующий раз}
    Notify := True;
  end;
end;

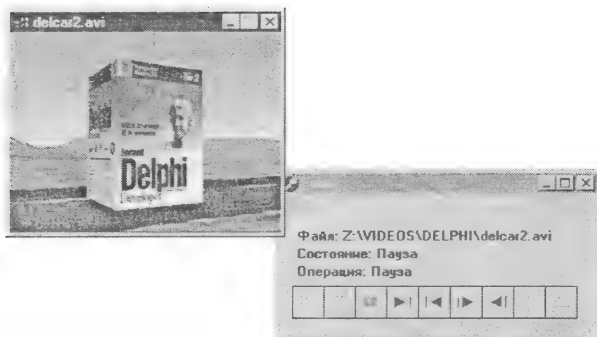
procedure TForm1.FormCreate(Sender: TObject);
begin
  with MediaPlayer1 do begin
    Label1.Caption:='Файл: '+FileName;
    Label2.Caption := 'Состояние: '+ModeStr[Mode];
    Notify:=true;
  end;
end;

procedure TForm1.MediaPlayer1Click(Sender: TObject;
  Button: TMPBtnType; var DoDefault:
  Boolean);
begin
  Label3.Caption:='Операция: '+ButtonStr[Button];
  {Переустановка Notify, чтобы произошло событие
  OnNotify}
  MediaPlayer1.Notify := True;
end;
end;

```

Рис. 4.25

Приложение универсального проигрывателя при воспроизведении видео файла



Запустите приложение и проверьте его в работе. На рис. 4.24 показано приложение в момент воспроизведения звукового файла. А на рис. 4.25 то же приложение воспроизводит видео файл. Причем, поскольку мы не определили окно и область вывода изображения, то проигрыватель сам создает соответствующее окно.

Глава 5

Кнопки, индикаторы, управляющие элементы

5.1 Общая характеристика

В данной главе будут рассмотрены такие управляющие элементы, как кнопки, индикаторы и некоторые другие. В таблице 5.1 приведен перечень этих элементов с краткими характеристиками. В этой таблице не указаны аналогичные элементы, связанные с базами данных, так как их рассмотрение выходит за рамки данной книги. Кнопка **UpDown** уже была рассмотрена в разделе 3.4.2.

На рис. 5.1 показаны примеры приведенных в таблице 5.1 компонентов.

Таблица 5.1. Кнопки, индикаторы и иные управляющие элементы

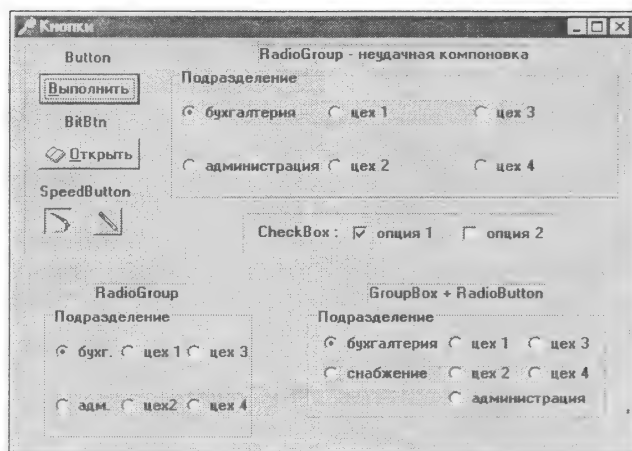
Пикто- грамма	Компонент	Страница	Описание
	Button (командная кнопка)	Standard	Используется для создания кнопок, которыми пользователь выполняет команды в приложении.
	BitBtn (кнопка с графикой)	Additional	Используется для создания кнопок, на которых располагается битовая графика (например, кнопка OK с галочкой).
	SpeedButton (кнопка с фиксацией и графикой)	Additional	Используется для создания инструментальных панелей кнопок с фиксацией нажатого состояния и в качестве быстрых кнопок, дублирующих команды меню.
	RadioGroup (группа радиокнопок)	Standard	Является комбинацией группового окна GroupBox с набором радиокнопок RadioButton ; служит специально для создания групп радиокнопок. Можно размещать в компоненте несколько радиокнопок, но никакие другие органы управления не разрешены.

Пикто- грамма	Компонент	Страница	Описание
	RadioButton (радиокнопка)	Standard	Предлагают пользователю набор альтернатив, из которого выбирается одна. Набор реализуется требуемым количеством радиокнопок, размещенных в одном контейнере (форме, панели и т. п.).
	GroupBox (групповое окно)	Standard	Является контейнером, объединяющим группу связанных органов управления, таких, как радиокнопки RadioButton , контрольные индикаторы Checkbox и т.д.
	UpDown (кнопка-счетчик)	Win32	Кнопка-счетчик в стиле Windows 95/98, в сочетании с компонентами Edit и другими позволяющая вводить цифровую информацию.
	Checkbox (контрольный индикатор с флажком)	Standard	Позволяет пользователю включать и выключать опции программы.
	CheckListBox (список с индикаторами)	Additional	Компонент является комбинацией свойств списка Listbox и индикаторов CheckBox в одном компоненте.
	TrackBar (ползунок)	System	Используется как элемент управления в виде ползунка.
	Scrollbar (линейка прокрутки)	Standard	Представляет собой стандартную линейку прокрутки Windows и служит для управления положением видимой части форм или компонентов.
	Timer (таймер)	Win32	Используется для запуска процедур, функций и событий в указанные интервалы времени.

В целом можно сказать, что в качестве управляющих кнопок используются обычно **Button** или, если желательно иметь на кнопке пиктограмму — **BitBtn**. Кнопки **SpeedButton** применяются в качестве быстрых кнопок и для построения инструментальных панелей, в которых требуется фиксация нажатого состояния. Компонент **RadioGroup** применяется для формирования группы регулярно размещенных радиокнопок, из которых в лю-

Рис. 5.1

Пример кнопок и индикаторов



бой момент времени может быть включена только одна. Если по различным соображениям регулярное размещение радиокнопок нежелательно, то используются компоненты **RadioButton**, размещенные желательным образом в панели **GroupBox**. Наконец, для введения в приложение различных опций, которые пользователь может включать и выключать, используются индикаторы **CheckBox**.

5.2 Управляющие кнопки **Button** и **BitBtn**

Простейшей и, пожалуй, наиболее часто используемой кнопкой является кнопка **Button** (см. на рис. 5.1 в верхнем левом углу формы), расположенная на странице библиотеки **Standard**. Реже используется кнопка **BitBtn** (см. на рис. 5.1 под кнопкой **Button**), отличающаяся, прежде всего, возможностью отобразить на ее поверхности изображение. Большинство свойств, методов и событий у этих видов кнопок одинаковы.

Основное с точки зрения внешнего вида свойство кнопки — **Caption** (надпись). В надписях кнопок можно предусматривать использование клавиш ускоренного доступа, выделяя для этого один из символов надписи. Перед символом, который должен соответствовать клавише ускоренного доступа, ставится символ амперсанта «&». Этот символ не появляется в надписи, а следующий за ним символ оказывается подчеркнутым. Тогда пользователь может вместо щелчка на кнопке нажать в любой момент клавишу **Alt** совместно с клавишей выделенного символа.

Например, если в вашем приложении имеется кнопка выполнения какой-то операции, вы можете задать ее свойство **Caption** равным «&Выполнить». На кнопке эта надпись будет иметь вид «Выполнить». И если пользователь нажмет клавиши Alt-B, то это будет эквивалентно щелчку на кнопке.

Основное событие любой кнопки — **OnClick**, возникающее при щелчке на ней. Именно в обработчике этого события записываются операторы, которые должны выполняться при щелчке пользователя на кнопке. Помимо этого есть еще ряд событий, связанных с различными манипуляциями клавишами и кнопками мыши.

Свойство **Cancel**, если его установить в **true**, определяет, что нажатие пользователем клавиши Esc будет эквивалентно нажатию на данную кнопку. Это свойство целесообразно задавать равным **true** для кнопок «Отменить» в различных диалоговых окнах, чтобы можно было выйти из диалога, нажав на эту кнопку или нажав клавишу Esc.

Свойство **Default**, если его установить в **true**, определяет, что нажатие пользователем клавиши ввода Enter будет эквивалентно нажатию на данную кнопку, даже если данная кнопка в этот момент не находится в фокусе. Правда, если в момент нажатия Enter в фокусе находится другая кнопка, то все-таки сработает именно кнопка в фокусе.

Еще одно свойство — **ModalResult** используется в модальных формах, рассмотрение которых выходит за рамки данной книги. В обычных приложениях значение этого свойства должно быть равно **mrNone**.

Из методов, присущих кнопкам, имеет смысл отметить один — **Click**. Выполнение этого метода эквивалентно щелчку на кнопке, т.е. вызывает событие кнопки **OnClick**. Этим можно воспользоваться, чтобы продублировать какими-то другими действиями пользователя щелчок на кнопке. Пусть, например, вы хотите, чтобы при нажатии пользователем клавиши с символом «С» или «с» в любой момент работы с приложением выполнялись операции, предусмотренные в обработчике события **OnClick** кнопки **Button1**. Поскольку неизвестно, какой компонент будет находиться в фокусе в момент этого события, надо перехватить его на уровне формы. Такой перехват осуществляется, если установить свойство формы **KeyPressPreview** в **true**. Тогда в обработчике события формы **OnKeyPress** можно написать оператор

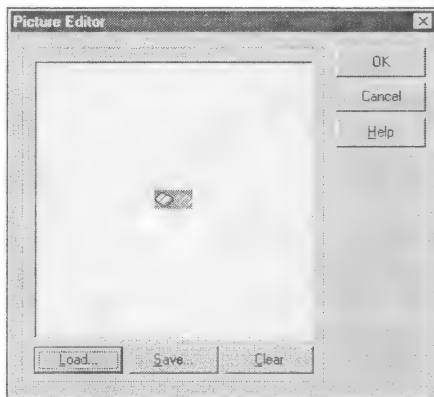
```
if (key='Y' or key='Z') then Button1.Click;
```

Если пользователь ввел символ «С» или «с», то в результате будет выполнен обработчик щелчка кнопки **Button1**.

Все сказанное выше в равной степени относится и к **Button**, и к **BitBtn**. Рассмотрим теперь особенности кнопки с пиктограммой **BitBtn**. Изображение на этой кнопке задается свойством **Glyph**. При нажатии кнопки с многоточием в строке свойства **Glyph** в Инспекторе Объектов вызывается окно, представленное на рис. 5.2. Нажав в нем кнопку **Load** вы перейдете в обычное окно открытия файла рисунка и можете выбрать файл битовой матрицы **.bmp**, содержащий желаемое изображение. В частности, с Delphi поставляется большое количество изображений для кнопок. Они расположены в каталоге `\Images\Buttons`, а сам каталог `Images` в Delphi 5 и 4 расположен в каталоге `\program files\common files\borland shared`, а в других версиях Delphi — в каталоге `\program files\borland\delphi...`.

Рис. 5.2

Окно редактора пиктограммы



После того, как вы выбрали изображение, нажмите **OK** и выбранное изображение появится на вашей кнопке левее надписи.

Файл изображения для кнопки может содержать до четырех изображений пиктограмм размера `16x16`. Самое левое соответствует отжатой кнопке. Второе слева соответствует недоступной кнопке, когда ее свойство **Enabled** равно **false**. Третье слева изображение используется при нажатии пользователя на кнопку при ее включении. Четвертое слева изображение используется в кнопках с фиксацией **SpeedButton**, о которых будет сказано позднее, для изображения кнопки в нажатом состоянии. Большинство изображений для кнопок использует две пиктограммы. Число пиктограмм вы можете узнать из свойства кнопки **NumGlyphs**, которое после загрузки изображения покажет вам число пиктограмм в нем.

Расположение изображения и надписи на кнопке определяется свойствами **Margin**, **Layout** и **Spacing**. Если свойство **Margin** равно -1 (значение по умолчанию), то изображение и надпись размещаются в центре кнопки. При этом положение изображения по отношению к надписи определяется свойством **Layout**, которое может принимать значения: **blGlyphLeft** (слева), **blGlyphRight** (справа), **blGlyphTop** (вверху), **blGlyphBottom** (внизу). Если же **Margin** > 0, то в зависимости от значения **Layout** изображение и надпись смещаются к той или иной кромке кнопки, отступая от нее на число пикселей, заданное значением **Margin**.

Свойство **Spacing** задает число пикселей, разделяющих изображение и надпись на поверхности кнопки. По умолчанию **Spacing** = 4. Если задать **Spacing** = 0, изображение и надпись будут размещены вплотную друг к другу. Если задать **Spacing** = -1, то текст появится посередине между изображением и краем кнопки.

Еще одно свойство **BitBtn** — свойство **Kind** определяет тип кнопки. По умолчанию значение этого свойства равно **bkCustom** — заказная. Но можно установить и множество других предопределенных типов: **bkOK**, **bkCancel**, **bkHelp**, **bkYes**, **bkNo**, **bkClose**, **bkAbort**, **bkRetry**, **bkIgnore**, **bkAll**. В этих типах уже сделаны соответствующие надписи, введены пиктограммы, заданы еще некоторые свойства. Обычно все-таки лучше ими не пользоваться. Во-первых, надписи все равно надо переводить на русский язык. Во-вторых, предопределенные рисунки обычно выбиваются из общего стиля конкретного приложения. И главное — предопределение некоторых свойств, не учтенных вами, может иногда приводить к странным результатам работы. Уж лучше использовать заказные кнопки и самому устанавливать в них все необходимые свойства.

5.3 Кнопка с фиксацией **SpeedButton**

Кнопки **SpeedButton** имеют возможность отображения пиктограмм и могут использоваться как обычные управляющие кнопки или как кнопки с фиксацией нажатого состояния (см. на рис. 5.1). Обычно они используются в качестве быстрых кнопок, дублирующих различные команды меню, и в инструментальных панелях, в которых требуется фиксация нажатого состояния.

У кнопок **SpeedButton**, как и у других кнопок, имеется свойство **Caption** — надпись, но в этих кнопках оно обычно оставляется пустым, так как вместо надписи используется пиктограмма.

Изображение на кнопке задается свойством **Glyph** точно так же, как описано в разделе 5.2 для кнопок **BitBtn**. И точно так же свойство **NumGlyphs** определяет число используемых пиктограмм, свойства **Layout** и **Margin** определяют расположение изображения, а свойство **Spacing** — расстояние между изображением и надписью (если, конечно, вы все-таки хотите использовать надпись на кнопке).

Особенностью кнопок **SpeedButton** являются свойства **GroupIndex** (индекс группы), **AllowAllUp** (разрешение отжатого состояния всех кнопок группы) и **Down** (исходное состояние — нажатое). Если **GroupIndex** = 0, то кнопка ведет себя так же, как **Button** и **BitBtn**. При нажатии пользователем кнопки она погружается, а при отпускании возвращается в нормальное состояние. В этом случае свойства **AllowAllUp** и **Down** не влияют на поведение кнопки.

Если **GroupIndex** > 0 и **AllowAllUp** = **true**, то кнопка при щелчке пользователя на ней погружается и остается в нажатом состоянии. При повторном щелчке пользователя на кнопке она освобождается и переходит в нормальное состояние (именно для того, чтобы освобождение кнопки состоялось, необходимо задать **AllowAllUp** = **true**). Если свойство **Down** во время проектирования установлено равным **true**, то исходное состояние кнопки — нажатое.

Если есть несколько кнопок, имеющих одинаковое ненулевое значение **GroupIndex**, то они образуют группу взаимосвязанных кнопок из которых нажатой может быть только одна. Если одна кнопка находится в нажатом состоянии и пользователь щелкает на другой, то первая кнопка освобождается, а вторая фиксируется в нажатом состоянии. Поведение нажатой кнопки при щелчке на ней зависит от значения свойства **AllowAllUp**. Если оно равно **true**, то кнопка освободится, поскольку в этом случае возможно состояние, когда все кнопки группы отжаты. Если же **AllowAllUp** равно **false**, то щелчок на нажатой кнопке не приведет к изменению вида кнопки. Впрочем, и в этом случае, как и при любом щелчке на кнопке, возникает событие **OnClick**, которое может быть обработано.

Состояние кнопки во время выполнения можно определить по значению свойства **Down**: если значение равно **true**, то кнопка нажата. Во время события **OnClick** значение **Down** уже равно тому состоянию, которое примет кнопка в результате щелчка на ней.

5.4 Группы радиокнопок — компоненты **RadioGroup**, **RadioButton** и **GroupBox**

Радиокнопки образуют группы взаимосвязанных индикаторов, из которых обычно может быть выбран только один. Они используются для выбора пользователем одной из нескольких взаимоисключающих альтернатив, например, отдела, в котором работает сотрудник, или пола сотрудника. Впрочем, радиокнопки могут использоваться и для отображения аналогичных данных. В этом случае управление кнопками осуществляется программно. Несколько примеров организации размещения радиокнопок вы можете увидеть на рис. 5.1 в разделе 5.1.

Начнем рассмотрение радиокнопок с компонента **RadioGroup** — панели группы радиокнопок. Это панель, которая может содержать регулярно расположенные столбцами и строками радиокнопки. Надпись в левом верхнем углу панели (см. рис. 5.1) определяется свойством **Caption**. А надписи кнопок и их количество определяются свойством **Items**, имеющим тип **TStrings**. Щелкнув на кнопке с многообразием около этого свойства в окне Инспектора Объектов, вы попадете в редактор списков строк, который уже рассматривался нами в разделе 3.3.4 (рис. 3.9). В нем вы можете занести надписи, которые хотите видеть около кнопок, по одной в строке. Сколько строчек вы запишете — столько и будет кнопок. Например, для компонента **RadioGroup** в верхней правой части формы рис. 5.1 свойство **Items** имеет вид:

```
бухгалтерия  
администрация  
цех 1  
цех 2  
цех 3  
цех 4
```

Кнопки, появившиеся в панели после задания значений **Items**, можно разместить в несколько столбцов (не более 17), задав свойство **Columns**. По умолчанию **Columns** = 1, т.е. кнопки размещаются друг под другом.

Определить, какую из кнопок выбрал пользователь, можно по свойству **ItemIndex**, которое показывает индекс выбранной кнопки. Индексы, как всегда в Delphi, начинаются с 0. По умолчанию **ItemIndex** = -1, что означает отсутствие выбранной кнопки. Если вы хотите, чтобы в момент начала выполнения приложения какая-то из кнопок была выбрана (это практически всегда необходимо), то надо установить соответствующее значение **ItemIndex** во время проектирования. Если вы используете радиокнопки не для ввода, а для отображения данных, устанавливать

значение **ItemIndex** можно программно во время выполнения приложения.

Компонент **RadioGroup** очень удобен, но не свободен от некоторых недостатков. Его хорошо использовать, если надписи кнопок имеют примерно одинаковую длину и если число кнопок в каждом столбце (при размещении их в нескольких столбцах) одинаково. Посмотрите, например, рис. 5.1 из раздела 5.1. Группа радиокнопок в нижнем левом углу формы имеет нормальный вид. А группа аналогичных радиокнопок в верхней правой части формы выглядит плохо: она занимает слишком много места, которое пропадает в пустую. Связано это с тем, что длина надписей у кнопок первого столбца превышает длину надписей у остальных кнопок. А **RadioGroup** при размещении кнопок ориентируется на надпись максимальной длины. Еще хуже выглядела бы эта группа, если бы число кнопок было, например, равно 5.

В подобных случаях желательно нерегулярное расположение кнопок. Такую возможность дают компоненты **RadioButton**, сгруппированные панелью **GroupBox**. Панель **GroupBox** выглядит на форме так же, как **RadioGroup** (см. рис. 5.1), и надпись в ее верхнем левом углу также определяется свойством **Caption**. Эта панель сама по себе пустая. Ее назначение — служить контейнером для других управляющих элементов, в частности, для радиокнопок **RadioButton**. Отдельная радиокнопка **RadioButton** особого смысла не имеет, хотя и может служить индикатором, включаемым и выключаемым пользователем. Но в качестве индикаторов обычно используются другие компоненты — **CheckBox**. А радиокнопки имеют смысл, когда они взаимодействуют друг с другом в группе. Эта группа и объединяется единым контейнером, обычно панелью **GroupBox**.

Рассмотрим свойства радиокнопки **RadioButton**. Свойство **Caption** содержит надпись, появляющуюся около кнопки. Значение свойства **Alignment** определяет, с какой стороны от кнопки появится надпись: **taLeftJustify** — слева, **taRightJustify** — справа (это значение принято по умолчанию). Свойство **Checked** определяет, выбрана ли данная кнопка пользователем, или нет. Поскольку в начале выполнения приложения обычно надо, чтобы одна из кнопок группы была выбрана по умолчанию, ее свойство **Checked** надо установить в **true** в процессе проектирования. Если вы поэкспериментируете, то заметите, что и во время проектирования можно установить в **true** значение **Checked** только у одной кнопки из группы.

Размещение кнопок **RadioButton** в панели **GroupBox**, как можно видеть из рис. 5.1 раздела 5.1, дает большую свободу по

сравнению с компонентом **RadioGroup** и позволяет разместить кнопки не регулярно.

Радиокнопки **RadioButton** могут размещаться не только в панели **GroupBox**, но и в любой панели другого типа, а также непосредственно на форме. Группа взаимосвязанных кнопок в этих случаях определяется тем оконным компонентом, который содержит кнопки. В частности, для радиокнопок, размещенных непосредственно на форме, контейнером является сама форма. Таким образом, все кнопки, размещенных непосредственно на форме, работают как единая группа, т.е. только в одной из этих кнопок можно установить значение **Checked** в **true**.

5.5 Индикаторы **CheckBox** и **CheckBox**

Индикаторы с флажком **CheckBox** (см. рис. 5.1 раздела 5.1) используются в приложениях в основном для того, чтобы пользователь мог включать и выключать какие-то опции, или для индикации состояния. При каждом щелчке пользователя на индикаторе его состояние изменяется, проходя в общем случае последовательно через три значения: выделение (появление черной галочки), промежуточное (серое окно индикатора и серая галочка) и не выделенное (пустое окно индикатора). Этим трем состояниям соответствуют три значения свойства компонента **State: cbChecked, cbGrayed, cbUnchecked**. Впрочем, эти три состояния допускаются только при значении другого свойства **AllowGrayed** равно **true**. Если же **AllowGrayed = false** (значение по умолчанию), то допускается только два состояния: выделенное и не выделенное. И **State**, и **AllowGrayed** можно устанавливать во время проектирования или программно во время выполнения.

Промежуточное состояние обычно используется, если индикатор применяется для отображения какой-то характеристики объекта. Например, если индикатор призван показать, какой регистр использовался при написании какого-то фрагмента текста, то в случае, если весь текст написан в верхнем регистре индикатор может принимать выделенное состояние, если в нижнем — не выделенное, а если использовались оба регистра — промежуточное.

Проверять состояние индикатора можно не только по значению **State**, но и по значению свойства **Checked**. Если **Checked** равно **true**, то индикатор выбран, т.е. **State = cbChecked**. Если **Checked** равно **false**, то **State** равно **cbUnchecked** или **cbGrayed**. Установка **Checked** в **true** во время проектирования или выполнения автоматически переключает **State** в **cbChecked**.

Как и в радиокнопке, в индикаторе **CheckBox** надпись задается свойством **Caption**, а ее размещение по отношению к индикатору — свойством **Alignment**.

Еще один компонент, имеющий индикаторы — список **CheckListBox**. Это аналог рассмотренного в разделе 3.3.5 компонента **ListBox**, но около каждой строки списка имеется индикатор, состояние которого пользователь может изменять. Вы можете увидеть, как выглядит список **CheckListBox**, на рис. 3.10 в разделе 3.3.5 (компонент расположен на рисунке в центре внизу).

Свойства, общие у **CheckListBox** и **Listbox**, мы рассматривать не будем, так как все, характеризующее этот компонент как список, рассмотрено в разделе 3.3.5. А состояния индикаторов определяют два свойства: **State** и **Checked**. Оба эти свойства можно рассматривать как индексированные массивы, каждый элемент которого соответствует индексу строки. Эти свойства можно устанавливать программно или читать, определяя установки пользователя. Например, операторы

```
CheckListBox1.Checked[1]:=true;  
CheckListBox1.State[2]:=cbGrayed;
```

устанавливают индикатор второй строки списка **CheckListBox1** в состояние выбранного, а индикатор третьей строки — в промежуточное состояние (вспомним, что индексы начинаются с 0).

Оператор

```
for i:=0 to CheckListBox1.Items.Count - 1 do  
  if CheckListBox1.Checked[i]  
  then ...
```

проверяет состояние всех индикаторов списка, и для выбранных пользователем строк осуществляет какие-то действия (в приведенном операторе на месте этих действий просто поставлено многоточие).

В компоненте **CheckListBox** имеется также событие **OnClickCheck**, возникающее при каждом изменении пользователем состояния индикатора. Его можно использовать для обработки результатов изменения.

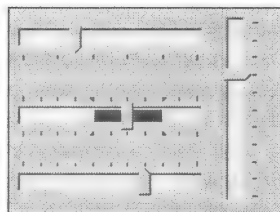
5.6 Ползунки и полосы прокрутки — компоненты **TrackBar** и **ScrollBar**

Компонент **TrackBar** представляет собой элемент управления в виде ползунка, который пользователь может перемещать курсором мыши или клавишами во время выполнения. Таким образом, пользователь может управлять какими-то процессами: громкостью звука, размером изображения и т.п. На рис. 5.3 при-

ведены различные формы отображения ползунка. Как видно из рисунка, он может располагаться горизонтально, вертикально, иметь шкалу с различных сторон, иметь какой-то выделенный диапазон шкалы.

Рис. 5.3

Различные варианты ползунков



Основное свойство компонента — **Position**. Это свойство можно задавать во время проектирования или программно во время выполнения. При перемещении пользователем ползунка можно прочитать значение **Position**, характеризующее позицию, в которую пользователь переместил ползунок. Для возможности такого чтения служит событие **OnChange**. В обработчике этого события можно прочитать значение **Position** и использовать его для управления каким-то компонентом.

Свойство **Position** — целое, значение которого может изменяться в пределах, задаваемых свойствами **Min** и **Max**. По умолчанию **Min** = 0, **Max** = 10, так что **Position** может принимать только 11 значений — от 0 до 10. Если задать большее значение **Max**, соответственно увеличится количество возможных значений **Position** в диапазоне **Min** — **Max**.

Свойство **Orientation** определяет ориентацию ползунка: **trHorizontal** — горизонтальная, **trVertical** — вертикальная.

Свойство **TickMarks** указывает размещение шкалы относительно компонента и может принимать значения: **tmBottomRight** — снизу или справа в зависимости от ориентации компонента (верхний и правый компоненты на рис. 5.3), **tmTopLeft** — сверху или слева в зависимости от ориентации компонента (нижний компонент на рис. 5.3), **tmBoth** — с обеих сторон (средний компонент на рис. 5.3).

Свойство **TickStyle** определяет способ изображения шкалы. Оно может принимать значения: **tsAuto** — автоматическая прорисовка шкалы, **tsNone** — отсутствие шкалы, **tsManual** — программное рисование шкалы с помощью метода **SetTick(Value: Integer)**, который помещает метку шкалы в позицию, соответствующую величине **Value**. Метки, соответствующие началу и концу шкалы автоматически размещаются и в случае **TickStyle** = **tsManual**.

При **TickStyle = tsAuto** частота меток шкалы определяется свойством **Frequency**. Это свойство задает, сколько возможных значений **Position** лежит между метками. Например, если **Frequency = 2**, то метки будут соответствовать только каждому второму возможному значению позиции (такое значение **Frequency** задано в верхнем компоненте на рис. 5.3).

Свойства **LineSize** и **PageSize** определяют, насколько смещается ползунок, если пользователь управляет им с помощью соответственно клавиш со стрелками или клавишами **PageUp** и **PageDown**.

Свойства **SelStart** и **SelEnd** позволяют визуально выделить на шкале некоторый диапазон (см. средний компонент на рис. 5.3), который о чем-то говорит пользователю, например, рекомендуемый диапазон значений. При этом ничто не мешает пользователю выйти за пределы этого диапазона.

Похож на ползунок по своим функциям и компонент **ScrollBar**, хотя выглядит он иначе и предназначен по замыслу для других целей. Этот компонент представляет собой стандартную линейку прокрутки **Windows**. Однако, он может использоваться и для целей прокрутки (впрочем, многие оконные компоненты **Delphi** имеют собственные полосы прокрутки), и для управления, подобного компоненту **TrackBar**.

Основные свойства **ScrollBar** — **Position**, **Min** и **Max** те же, что у компонента **TrackBar**. Свойство **Kind**, определяющее горизонтальное или вертикальное расположение полосы и принимающее соответственно значения **sbHorizontal** или **sbVertical**, аналогично свойству **Orientation** компонента **TrackBar**.

Имеются два свойства, отсутствующие у **TrackBar**: **SmallChange** и **LargeChange**. Они определяют соответственно «малый» сдвиг при щелчке на кнопке в конце полосы или нажатии клавиши со стрелкой, и «большой» сдвиг при перемещении на страницу щелчком рядом с бегунком или нажатием клавиш **PageUp** или **PageDown**.

Событие, соответствующее перемещению пользователем бегунка полосы прокрутки — **OnScroll**. В процедуру обработчика этого события передается по ссылке параметр **ScrollPos** — позиция бегунка, которую можно читать, но можно и изменять, и передается параметр **ScrollCode**, характеризующий вид перемещения бегунка. Этот параметр может иметь значения:

scLineUp, scLineDown	«Малый» сдвиг: перемещение соответственно вверх или налево и вниз или вправо после нажатия кнопки полосы прокрутки или клавиши со стрелкой
scPageUp, scPageDown	«Большой» сдвиг: перемещение на страницу щелчком рядом с бегунком или нажатием клавиш PageUp или PageDown
scPosition	Пользователь переместил и освободил бегунок
scTrack	Пользователь перемещает бегунок
scTop, scBottom	Бегунок перемещен соответственно в крайнюю верхнюю или левую позицию и в крайнюю нижнюю или правую позицию
scEndScroll	Окончание перемещения

В обработке события **ScrollPos** можно поместить операторы, перемещающие требуемую область формы или компонент, а можно поместить операторы, которые управляют некоторым компонентом, используя значение позиции бегунка **ScrollPos**.

5.7 Таймер — компонент **Timer**

Компонент **Timer** позволяет задавать в приложении интервалы времени. Таймер находит многочисленные применения: синхронизация мультипликации, закрытие каких-то окон, с которыми пользователь долгое время не работает, включение хранителя экрана или закрытие связей с удаленным сервером при отсутствии действий пользователя, регулярный опрос каких-то источников информации, задание времени на ответ в обучающих программах — все это множество задач, в которых требуется задавать интервалы времени, решается с помощью таймера.

Таймер — невизуальный компонент, который может размещаться в любом месте формы. Он имеет два свойства, позволяющие им управлять: **Interval** — интервал времени в миллисекундах и **Enabled** — доступность. Свойство **Interval** задает период срабатывания таймера. Через заданный интервал времени после предыдущего срабатывания, или после программной установки свойства **Interval**, или после запуска приложения, если значение **Interval** установлено во время проектирования, таймер срабатывает, вызывая событие **OnTimer**. В обработчике этого события записываются необходимые операции.

Если задать **Interval = 0** или **Enabled = false**, то таймер перестает работать. Чтобы запустить отсчет времени надо или задать

Enabled = true, если установлено положительное значение **Interval**, или задать положительное значение **Interval**, если **Enabled = false**.

Например, если требуется, чтобы через 5 секунд после запуска приложения закрылась форма — заставка, отображающая логотип приложения, на ней надо разместить таймер, задать в нем интервал **Interval = 5000**, а в обработчик события **OnTimer** вставить оператор **Close**, закрывающий окно формы.

Если необходимо в некоторой процедуре запустить таймер, который отсчитал бы заданный интервал, например, 5 секунд, после чего надо выполнить некоторые операции и отключить таймер, это можно сделать следующим образом. При проектировании таймер делается доступным (**Enabled = true**), но свойство **Interval** задается равным 0. Таймер не будет работать, пока в момент, когда нужно запустить таймер, не выполнится оператор

```
Timer1.Interval:=5000;
```

Через 5 секунд после этого наступит событие **OnTimer**. В его обработчике надо задать оператор

```
Timer1.Interval:=0;
```

который отключит таймер, после чего можно выполнять требуемые операции.

Другой эквивалентный способ решения задачи — использование свойства **Enabled**. В время проектирования задается значение **Interval = 5000** и значение **Enabled = false**. В момент, когда надо запустить таймер выполняется оператор

```
Timer1.Enabled:=true;
```

В обработчик события **OnTimer**, которое наступит через 5 секунд после запуска таймера, можно вставить оператор

```
Timer1.Enabled:=false;
```

который отключит таймер.

Таймер точно выдерживает заданные интервалы **Interval**, если они достаточно велики — сотни и тысячи миллисекунд. Если же задавать интервалы длительностью десятки или единицы миллисекунд, то реальные интервалы времени оказываются заметно больше вследствие различных накладных расходов, связанных с вызовами функций и иными вычислительными аспектами.

1870-1871. The first year of the war.

1872-1873. The second year of the war.

1874-1875. The third year of the war.

1876-1877. The fourth year of the war.

1878-1879. The fifth year of the war.

1880-1881. The sixth year of the war.

1882-1883. The seventh year of the war.

1884-1885. The eighth year of the war.

1886-1887. The ninth year of the war.

1888-1889. The tenth year of the war.

1890-1891. The eleventh year of the war.

1892-1893. The twelfth year of the war.

1894-1895. The thirteenth year of the war.

1896-1897. The fourteenth year of the war.

1898-1899. The fifteenth year of the war.

1900-1901. The sixteenth year of the war.

1902-1903. The seventeenth year of the war.

1904-1905. The eighteenth year of the war.

6.1 Главное меню — компонент MainMenu

В Delphi имеется два компонента, представляющие меню: **MainMenu** — главное меню, и **PopupMenu** — всплывающее меню. Оба компонента расположены на странице Standard. Эти компоненты имеют много общего. Начнем рассмотрение с компонента **MainMenu**.

Это невидимый компонент, т.е. место его размещения на форме в процессе проектирования не имеет никакого значения для пользователя — он все равно увидит не сам компонент, а только меню, сгенерированное им.

Обычно на форму помещается один компонент **MainMenu**. В этом случае его имя автоматически заносится в свойство формы **Menu**. Но можно поместить на форму и несколько компонентов **MainMenu** с разными наборами разделов, соответствующими различным режимам работы приложения. В этом случае во время проектирования свойству **Menu** формы присваивается ссылка на один из этих компонентов. А в процессе выполнения в нужные моменты это свойство можно изменять, меняя соответственно состав главного меню приложения.

Основное свойство компонента — **Items**. Его заполнение производится с помощью Конструктора Меню, вызываемого двойным щелчком на компоненте **MainMenu** или нажатием кнопки с многоточием рядом со свойством **Items** в окне Инспектора Объектов. В результате откроется окно, вид которого представлен на

Рис. 6.1
Окно Конструктора
Меню

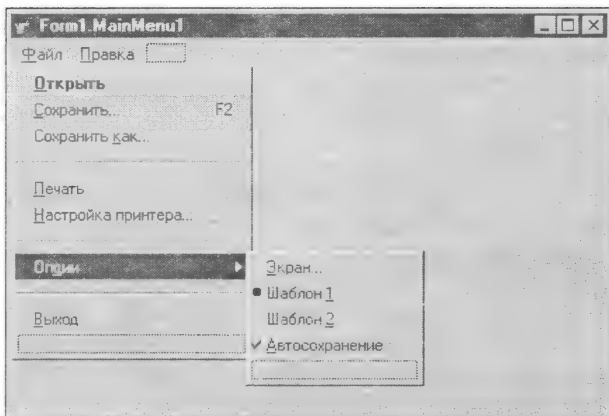


Рис. 6.2

Результат конструирования меню

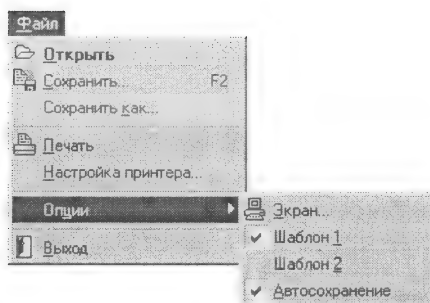


рис. 6.1. В этом окне вы можете спроектировать все меню. На рис. 6.2 показано в работе то меню, которое соответствует проектируемому на рис. 6.1.

При работе в конструкторе меню новые разделы можно вводить, помещая курсор в рамку из точек, обозначающую место расположения нового раздела (см. рис. 6.1). Если при этом раздел ввелся не на нужном вам месте, вы можете отбуксировать его мышью туда, куда вам надо. Другой путь ввода нового раздела — использование контекстного меню, всплывающего при щелчке правой кнопкой мыши. Если вы предварительно выделите какой-то раздел меню и выберите из контекстного меню команду *Insert*, то рамка нового раздела вставится перед ранее выделенным. Из контекстного меню вы можете также выполнить команду *Create Submenu*, позволяющую ввести подменю в выделенный раздел (см. подменю раздела *Опции* на рис. 6.1, 6.2).

При выборе нового раздела вы увидите в Инспекторе Объектов множество свойств данного раздела. Дело в том, что каждый раздел меню, т.е. каждый элемент свойства **Items**, является объектом типа **TMenuItem**, обладающим своими свойствами, методами, событиями.

Свойство **Caption** обозначает надпись раздела. Заполнение этого свойства подчиняется тем же правилам, что и заполнение аналогичного свойства в кнопках (см. раздел 5.2), включая использование символа амперсанта для обозначения клавиш быстрого доступа. Если вы в качестве значения **Caption** очередного раздела введете символ минус «-», то вместо раздела в меню появится разделитель (см. на рис. 6.1 и 6.2 разделители после разделов *Сохранить как*, *Настройка принтера* и *Опции*).

Свойство **Name** задает имя объекта, соответствующего разделу меню. Очень полезно давать этим объектам осмысленные имена, так как иначе вы скоро запутаетесь в ничего не говорящих именах типа **N21**. Куда понятнее имена типа **MFile**, **MOpen**, **MSave** и т.п.

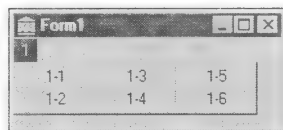
Свойство **ShortCut** определяет клавиши быстрого доступа к разделу меню — «горячие» клавиши, с помощью которых пользователь, даже не заходя в меню, может в любой момент вызвать выполнение процедуры, связанной с данным разделом. Чтобы определить клавиши быстрого доступа, надо открыть выпадающий список свойства **ShortCut** в окне Инспектора Объектов и выбрать из него нужную комбинацию клавиш. Эта комбинация появится в строке раздела меню (см. команду Сохранить... на рис. 6.1, 6.2). В разделе 6.3 рассказано о некоторых дополнительных возможностях задания комбинаций горячих клавиш.

Свойство **Default** определяет, является ли данный раздел разделом по умолчанию своего подменю, т.е. разделом, выполняемым при двойном щелчке пользователя на родительском разделе. Подменю может содержать только один раздел по умолчанию, выделяемый жирным шрифтом (см. раздел Открыть на рис. 6.1, 6.2).

Свойство **Break** используется в длинных меню, чтобы разбить список разделов на несколько столбцов. Возможные значения **Break**: **mbNone** — отсутствие разбиения меню (это значение принято по умолчанию), **mbBarBreak** и **mbBreak** — в меню вводится новый столбец разделов, отделенный от предыдущего полосой (**mbBarBreak**) или пробелами (**mbBreak**). На рис. 6.3 показан пример, в котором в разделе 1-3 установлено значение **Break = mbBreak**, а в разделе 1-5 — **Break = mbBarBreak**.

Рис. 6.3

Пример меню с разбиением на столбцы



Свойство **Checked**, установленное в **true**, указывает, что в разделе меню будет отображаться маркер флажка, показывающий, что данный раздел выбран (см. на рис. 6.1, 6.2 раздел «Автосохранение»). Правда, сам по себе этот маркер не изменяется и в обработчик события **OnClick** такого раздела надо вставлять оператор типа

```
MAutoSave.Checked:= not MAutoSave.Checked;
```

(в приведенном операторе подразумевается, что раздел меню назван **MAutoSave**).

Еще одним свойством, позволяющим вводить маркеры в разделы меню, является **RadioItem**. Это свойство, установленное в **true**, определяет, что данный раздел должен работать в режиме радиокнопки совместно с другими разделами, имеющими то же

значение свойства **GroupIndex**. По умолчанию значение **GroupIndex** равно 0. Но можно задать его большим нуля и тогда, если имеется несколько разделов с одинаковым значением **GroupIndex** и с **RadioItem = true**, то в них могут появляться маркеры флажков, причем только в одном из них (на рис. 6.1, 6.2 свойство **RadioItem** установлено в **true** в разделах Шаблон 1 и Шаблон 2, имеющих одинаковое значение **GroupIndex**). Если вы зададите программно в одном из этих разделов **Checked = true**, то в остальных разделах **Checked** автоматически сбросится в **false**. Впрочем, установка **Checked = true** лежит на программе; эта установка может выполняться аналогично приведенному выше оператору.

Описанные маркеры флажков в режиме радиокнопок и в обычном режиме используются для разделов меню, представляющих собой различные опции, взаимоисключающие или совместимые.

Для каждого раздела могут быть установлены во время проектирования или программно во время выполнения свойства **Enabled** (доступен) и **Visible** (видимый). Если установить **Enabled = false**, то раздел будет изображаться серой надписью и не будет реагировать на щелчок пользователя. Если же задать **Visible = false**, то раздел вообще не будет виден, а остальные разделы сомкнутся, заняв место невидимого. Свойства **Enabled** и **Visible** используются для того, чтобы изменять состав доступных пользователю разделов в зависимости от режима работы приложения.

Начиная с Delphi 4 предусмотрена возможность ввода в разделы меню изображений. За это ответственны свойства разделов **Bitmap** и **ImageIndex**. Первое из них позволяет непосредственно ввести изображение в раздел, выбрав его из указанного вами файла. Второе позволяет указать индекс изображения, хранящегося во внешнем компоненте **ImageList** (см. раздел 9.3). Указание на этот компонент вы можете задать в свойстве **Images** компонента **MainMenu**. Индексы начинаются с 0. Если вы укажете индекс -1 (значение по умолчанию), изображения не будет.

Мы рассмотрели все основные свойства объектов, соответствующих разделам меню. Основное событие раздела — **OnClick**, возникающее при щелчке пользователя на разделе или при нажатии «горячих» клавиш быстрого доступа.

Рассмотрим теперь вопросы объединения главных меню вторичных форм с меню главной формы. Речь идет о приложениях с несколькими формами, в которых и главная, и вспомогательные формы имеют свои главные меню — компоненты **MainMenu**. Конечно, пользователю неудобно работать одновременно с несколь-

кими окнами, каждое из которых имеет свое меню. Обычно надо, чтобы эти меню сливались в одно меню главной формы.

Приложения с несколькими формами могут быть двух видов: приложения с интерфейсом множества документов — так называемые MDI приложения, и обычные приложения с главной и вспомогательными формами. Типичными примерами приложений MDI являются программы Word и Excel. Рассмотрение особенностей этих видов приложений выходит за рамки данной книги. Сейчас нас интересует только один вопрос: как объединяются меню различных форм. В MDI приложениях меню дочерних форм всегда объединяются с меню родительской формы. А в приложениях с несколькими формами наличие или отсутствие объединения определяется свойством **AutoMerge** компонентов **TMainMenu**. Если требуется, чтобы меню вторичных форм объединялись с меню главной формы, то в каждой такой вторичной форме надо установить **AutoMerge** в **true**. При этом свойство **AutoMerge** главной формы должно оставаться в **false**.

Способ объединения меню определяется свойством разделов **GroupIndex**. По умолчанию все разделы меню имеют одинаковое значение **GroupIndex**, равное нулю. Если требуется объединение меню, то разделам надо задать неубывающие номера свойств **GroupIndex**. Тогда, если разделы встраиваемого меню имеют те же значения **GroupIndex**, что и какие-то разделы меню основной формы, то эти разделы заменяют соответствующие разделы основного меню. В противном случае разделы вспомогательного меню встраиваются между элементами основного меню в соответствии с номерами **GroupIndex**. Если встраиваемый раздел имеет **GroupIndex** меньший, чем любой из разделов основного меню, то разделы встраиваются в начало.

Пусть, например, в основной и вторичной формах структуры меню имеет следующие значения **GroupIndex**:

Форма 1	Форма 2
2 — 4	1 — 3
2 4	1 3
2 4	1

Тогда в момент, когда активизируется вторая форма, в первой появляется меню со структурой:

1 — 2 — 3 — 4
1 2 3 4
1 2 4

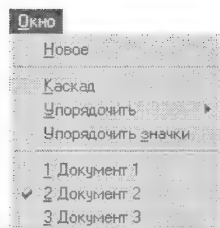
В этом примере отсутствовали разделы, имеющие в обеих формах одинаковые значения **GroupIndex**. Если бы такие были, то при активизации второй формы соответствующие разделы ее меню заменили бы аналогичные разделы первой формы.

Если в меню имеются разделы, работающие как радиокнопки, то нельзя забывать, что их взаимодействие также определяется свойствами **GroupIndex**.

Теперь остановимся на одном из вопросов, связанных с меню в упоминавшихся выше приложениях MDI. В них пользователь может открывать сколько ему требуется окон документов. Обычно в подобных приложениях имеется меню Окно (см. рис. 6.4), которое содержит такие разделы, как Новое, Упорядочить и т.п. Последним идет обычно список открытых окон документов, в который заносятся названия открытых пользователем окон. Выбирая в этом списке, пользователь может переключаться между окнами документов.

Рис. 6.4

Меню «Окно» в приложении MDI со списком открытых документов

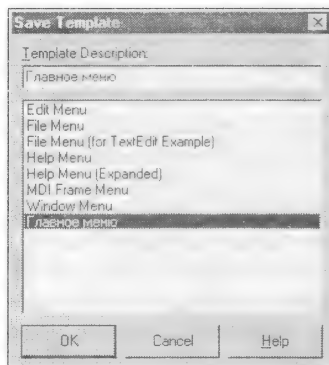


Для включения в меню раздела списка открытых окон, надо в свойстве WindowMenu главной формы приложения MDI указать имя меню, в конец которого должен помещаться список. Указывается именно имя меню, а не разделов выпадающего списка. Для примера рис. 6.4 должно быть указано имя элемента меню, соответствующего команде Окно.

Одним из безусловных требований, предъявляемых к меню приложений для Windows, является стандартизация меню и их разделов. Этому помогает команда Save As Template... в контекстном меню, всплывающем при щелчке правой кнопкой мыши в окне Конструктора Меню. Эта команда вызывает диалог, представленный на рис. 6.5. В этом диалоге вы можете в верхнем окне указать описание (заголовок), под которым хотите сохранить ваше меню. Впоследствии в любом вашем новом приложении вы можете загрузить этот шаблон в меню, выбирая из всплывающего меню в окне Конструктора Меню команду Insert From Template....

Рис. 6.5

Окно сохранения шаблона разработанного меню



На этом мы пока закончим рассмотрение компонента **MainMenu**. В разделе 6.3 мы еще вернемся к нему и покажем на примере один из видов настройки меню в процессе выполнения приложения.

6.2 Контекстное всплывающее меню — компонент **PopupMenu**

Контекстное меню привязано к конкретным компонентам. Оно всплывает, если во время, когда данный компонент в фокусе, пользователь щелкнет правой кнопкой мыши. Обычно в контекстное меню включают те команды главного меню, которые в первую очередь могут потребоваться при работе с данным компонентом.

Контекстному меню соответствует компонент **PopupMenu**. Поскольку в приложении может быть несколько контекстных меню, то и компонентов **PopupMenu** может быть несколько. Оконные компоненты: панели, окна редактирования, а также метки и др. имеют свойство **PopupMenu**, которое по умолчанию пусто, но куда можно поместить имя того компонента **PopupMenu**, с которым будет связан данный компонент.

Формирование контекстного всплывающего меню производится с помощью Конструктора Меню, вызываемого двойным щелчком на **PopupMenu**, точно так же, как это делалось для главного меню. Обратим только внимание на возможность упрощения этой работы. Поскольку разделы контекстного меню обычно повторяют некоторые разделы уже сформированного главного меню, то можно обойтись копированием соответствующих разделов. Для этого, войдя в Конструктор Меню из компонента **PopupMenu**, щелкните правой кнопкой мыши и из

всплывшего меню выберите команду **Select Menu** (выбрать меню). Вам будет предложено диалоговое окно, в котором вы можете перейти в главное меню. В нем вы можете выделить нужный вам раздел или разделы (при нажатой клавише **Shift** выделяются разделы в заданном диапазоне, при нажатой клавише **Ctrl** можно выделить совокупность разделов, не являющихся соседними). Затем выполните копирование их в буфер обмена, нажав клавиши **Ctrl-C**. После этого опять щелкните правой кнопкой мыши, выберите команду **Select Menu** и вернитесь в контекстное меню. Укажите курсором место, в которое хотите вставить скопированные разделы, и нажмите клавиши чтения из буфера обмена — **Ctrl-V**. Разделы меню вместе со всеми их свойствами будут скопированы в создаваемое вами контекстное меню.

В остальном работа с **PopupMenu** не отличается от работы с **MainMenu**. Только не возникает вопросов объединения меню разных форм: контекстные меню не объединяются.

6.3 Горячие клавиши — компонент **HotKey**

Компонент **HotKey**, расположенный в библиотеке на странице **Win32**, является вспомогательным, обеспечивающим возможность задания самим пользователем горячих клавиш, определяющих быстрый доступ к разделам меню. К тому же этот компонент позволяет задать такие сочетания горячих клавиш, которые не предусмотрены в выпадающем списке свойства разделов меню **ShortCut**.

Компонент **HotKey** внешне выглядит как обычное окно редактирования **Edit**. Но если в него входит пользователь, то оно переводит нажимаемые им клавиши в тип **TShortCut**, хранящий комбинацию горячих клавиш. Например, если пользователь нажимает клавиши **Ctrl-ф**, то в окне **HotKey** появится текст «**Ctrl + ф**».

Основное свойство компонента — **HotKey**, равное по умолчанию комбинации клавиш **Alt-A**. Это свойство можно прочесть и присвоить свойству **ShortCut** какого-то раздела меню. Например, оператор

```
MOpen.ShortCut:=HotKey1.HotKey;
```

присваивает разделу меню с именем **MOpen** комбинацию клавиш, заданную в компоненте **HotKey1**.

Свойство **Modifiers** указывает модификатор — вспомогательную клавишу, нажимаемую перед символьной. Это свойство является множеством, которое может включать значения **hkShift**, **hkCtrl**, **hkAlt**, **hkExt**, что соответствует клавишам **Shift**, **Ctrl**, **Alt**, **Extra**. По умолчанию **Modifiers** = [**hkAlt**]. Если вы хотите, напри-

мер, задать вместо этого значения в качестве модификатора клавишу Ctrl, вы должны выполнить оператор:

```
HotKey1.Modifiers:=[hkCtrl];
```

Свойство **InvalidKeys** задает недопустимые клавиши или их комбинации. Это свойство является множеством, которое может включать значения **hcNone**, **hcShift**, **hcCtrl**, **hcAlt**, **hcShiftCtrl**, **hcShiftAlt**, **hcCtrlAlt**, **hcShiftCtrlAlt**, что соответствует отсутствию модификатора и клавишам Shift, Ctrl, Alt, Shift-Ctrl, Shift-Alt, Ctrl-Alt, Shift-Ctrl-Alt.

Если вы хотите задать программно значение свойства **HotKey**, то можете это сделать, например, операторами

```
HotKey1.HotKey:=ord('F');  
HotKey1.Modifiers:=[hkAlt];
```

Эти операторы зададут комбинацию горячих клавиш Alt-F.

В заключение приведем пример использования компонента **HotKey** и настройки горячих клавиш меню в процессе выполнения приложения.

Пусть у вас есть главная форма приложения, содержащая компонент **MainMenu** и пусть вы хотите ввести команду настройки, позволяющую пользователю изменить установленные для разделов меню горячие клавиши. Для упрощения задачи будем считать, что меню, сконструированное в **MainMenu**:

- не каскадное (т.е. состоит только из двух уровней — заголовков меню и выпадающих списков разделов)
- в свойствах **Caption** разделов меню не использованы амперсаны
- в меню отсутствуют разделители

Эти предположения сделаны просто для того, чтобы упростить код и не использовать функции, не описанные в данной книге.

Начните новое приложение, разместите на форме компонент **MainMenu** и сконструируйте с его помощью любое меню, удовлетворяющее перечисленным требованиям. Задайте каким-то из разделов меню быстрые клавиши. Один из разделов меню должен называться Настройка и при выборе его мы хотим предоставить пользователю вспомогательную форму для настройки быстрых клавиш.

Добавьте в приложение еще одну форму (команда File | New Form). Эта форма будет вспомогательной. В обработчик команды Настройка главной формы вставьте оператор

```
Form2.ShowModal;
```

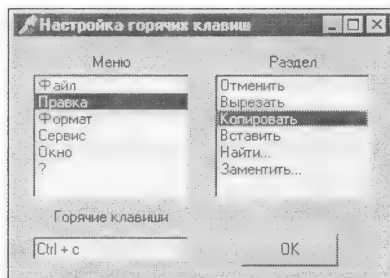
Этот оператор покажет пользователю окно вспомогательной формы как модальное — т.е. пользователь не сможет вернуться в

главную форму, пока не закроет вспомогательную. Чтобы компилятор понял этот оператор, надо в модуль главной формы **Unit1** вставить оператор **uses**, ссылающийся на модуль вспомогательной формы **Unit2**. Можете сделать это вручную или, перейдя в окне Редактора Кода в модуль **Unit1** выполните команду **File | Use Unit** и укажите, что хотите связаться с модулем **Unit2**. Поскольку из модуля **Unit2** надо будет видеть меню модуля **Unit1**, то аналогичным образом введите и обратную связь — свяжите модуль **Unit2** с **Unit1**.

Теперь давайте спроектируем вспомогательную форму. Она может иметь вид, представленный на рис. 6.6. На ней расположено два списка **ListBox** (см. раздел 3.3.5): **ListBox1**, в котором отображаются заголовки меню, и **ListBox2**, в котором отображаются разделы меню, соответствующие выбранному заголовку. В нижней части формы расположен компонент **HotKey** и кнопка **Button**, которая фиксирует в меню сделанный пользователем выбор и закрывает форму. В компоненте **HotKey** надо стереть с помощью Инспектора Объектов свойство **HotKey**, которое содержит некоторое значение по умолчанию.

Рис. 6.6

Окно настройки горячих клавиш во время выполнения



Теперь надо написать обработчики событий, которые загружали бы списки **ListBox1** и **ListBox2** названиями разделов и при выборе пользователем быстрых клавиш фиксировали бы этот выбор в соответствующем разделе меню формы **Form1**. Код может выглядеть следующим образом:

```
procedure TForm2.FormShow(Sender: TObject);
var i:integer;
begin
{ Загрузка ListBox1 заголовками меню
  при событии OnShow формы Form2 }
  ListBox1.Clear;
  for i:=0 to Form1.MainMenu1.Items.Count-1 do
    ListBox1.Items.Add(Form1.MainMenu1.Items[i].Caption);
  ListBox1.ItemIndex:=0;
```

```
{ Обращение к процедуре загрузки ListBox2 }
  ListBox1Click(Sender);
end;

procedure TForm2.ListBox1Click(Sender: TObject);
var i:integer;
begin
  { Загрузка ListBox2 заголовками разделов меню
    MainMenu1.Items[ListBox1.ItemIndex],
    выделенного пользователем в ListBox1
    при событии OnShow формы Form2 }
  ListBox2.Clear;
  for i:=0 to
Form1.MainMenu1.Items[ListBox1.ItemIndex].Count-1 do
    ListBox2.Items.Add(Form1.MainMenu1.Items[
                        ListBox1.ItemIndex].Items[i].Caption);
  ListBox2.ItemIndex:=0;
end;

procedure TForm2.ListBox2Click(Sender: TObject);
begin
  { Занесение горячих клавиш выделенного в ListBox2
    раздела в компонент HotKey1 }
  HotKey1.HotKey:=Form1.MainMenu1.Items[
ListBox1.ItemIndex].Items[ListBox2.ItemIndex].ShortCut;
end;

procedure TForm2.Button1Click(Sender: TObject);
begin
  { Изменение горячих клавиш выбранного раздела меню
    и закрытие вспомогательной формы }
  Form1.MainMenu1.Items[
    ListBox1.ItemIndex].Items[
    ListBox2.ItemIndex].ShortCut:=HotKey1.HotKey;
  Close;
end;
```

При событии **OnShow** формы **Form2** происходит загрузка списка **ListBox1** заголовками меню. Цикл загрузки перебирает индексы от 0 до **Form1.MainMenu1.Items.Count-1**. Это значение на 1 меньше значения **MainMenu1.Items.Count**, которое равно числу элементов в свойстве **MainMenu1.Items**.

При щелчке пользователя на списке **ListBox1** происходит загрузка списка **ListBox2**. При этом к соответствующим разделам меню получается доступ с помощью выражения **Form1.MainMenu1.Items[ListBox1.ItemIndex].Items[i]**. В этом выражении **Form1.MainMenu1.Items[ListBox1.ItemIndex]** — элемент головного раздела меню, выбранного пользователем в **ListBox1**. Каждый такой раздел можно рассматривать как элемент массива

меню и в то же время он сам является массивом разделов второго уровня. Поэтому его свойство **Items[i]** указывает на подраздел с индексом **i**.

При щелчке пользователя на списке **ListBox2** происходит загрузка компонента **HotKey1** символами горячих клавиш выбранного пользователем раздела. Если раздел не имеет горячих клавиш, то в окне **HotKey1** отображается текст «Нет». Далее пользователь может войти в окно **HotKey1** и нажать сочетание клавиш, которое он хочет назначить выбранному им разделу меню. Обработка щелчка на кнопке фиксирует это сочетание в разделе меню и закрывает вспомогательную форму.

Опробуйте это приложение в работе и вам станет яснее механизм работы с разделами меню и с быстрыми клавишами.

7.1 Общая характеристика

Панели являются контейнерами, служащими для объединения других управляющих элементов. Они могут выполнять как чисто декоративные функции, зрительно объединяя компоненты, связанные друг с другом по назначению, так и функции управления, организуя совместную работу своих дочерних компонентов. Примером этого была рассмотренная в разделе 5.4 организация работы группы радиокнопок.

В таблице 7.1 приведен перечень панелей, обслуживающих их компонентов и компонентов внешнего оформления, включенных в библиотеку Delphi 5.

Таблица 7.1 Панели и обслуживающие их компоненты

Пикто- грамма	Компонент	Страница	Описание
	GroupBox (групповое окно)	Standard	Является контейнером, объединяющим группу связанных органов управления, таких, как радиокнопки RadioButton , контрольные индикаторы Checkbox и т.д.
	Panel (панель)	Standard	Является контейнером для группирования органов управления и меньших контейнеров. Панель можно использовать также для построения полос состояния, инструментальных панелей, палитр инструментов.
	Bevel (рамка)	Additional	Используется для рисования прямоугольной рамки, изображенной как выступающая или утопленная.
	ScrollBox (окно с про- круткой)	Additional	Используется для создания зон отображения с прокруткой.
	Splitter (разделитель панелей)	Additional	Используется для создания в приложении панелей с изменяемыми пользователем размерами.

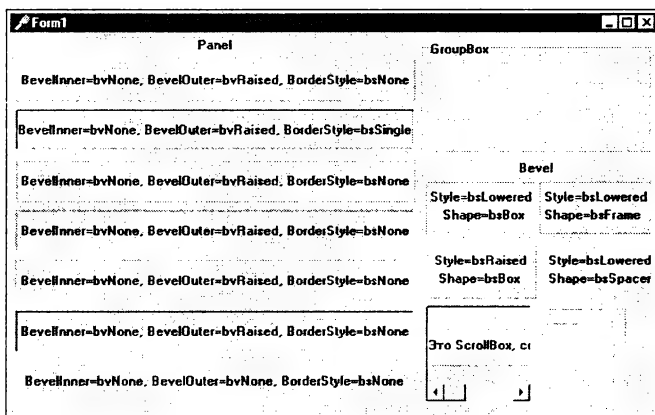
Пикто- грамма	Компонент	Страница	Описание
	ControlBar (инструменталь- ная панель)	Additional	Используется для размещения компонентов инструментальной панели.
	TabControl (страница с за- кладкой)	Win32	Позволяет добавлять закладки в стиле Windows 95/98, которые может выбирать пользователь.
	PageControl (многостранич- ное окно)	Win32	Позволяет создавать страницы в стиле Windows 95/98, управляемые закладками или иными органами управления, для экономии места на рабочем столе.
	StatusBar (полоса состоя- ния)	Win32	Полоса состояния приложения, при необходимости — на нескольких панелях.
	ToolBar (инструмента- льная панель)	Win32	Инструментальная панель для быстрого доступа к часто используемым функциям приложения.
	CoolBar (инструменталь- ная перестраи- ваемая панель)	Win32	Контейнер инструментальной панели, размеры которой могут изменяться пользователем.
	PageScroller (прокрутка страниц)	Win32	Обеспечивает прокрутку больших окон, например, инструментальных панелей.
	TabSet (блокнот с за- кладками)	Win3.1	Используется для создания блокнота с закладками.
	TabbedNoteBo- ok (многостра- ничная форма)	Win3.1	Используется для создания многостраничных форм с закладками.
	NoteBook (пачка стра- ниц)	Win3.1	Используется для создания пачки страниц, может применяться совместно с TabSet .
	Frame (фрейм)	Standard	Используется как проектируе- мый в виде отдельного окна контейнер любых компонентов. Обладает возможностями насле- дования, может включаться в Депозитарий.

7.2 Панели общего назначения — компоненты Panel, GroupBox, Bevel, ScrollBox, Splitter

На рис. 7.1 приведен пример, демонстрирующий вид таких панелей, как **Panel**, **GroupBox**, **Bevel**, **ScrollBox**. В левой части формы размещены компоненты **Panel** с различными значениями параметров. С этих компонентов мы и начнем рассмотрение панелей.

Рис. 7.1

Пример панелей общего назначения



Панели **Panel** используются наиболее широко. С их помощью komponуются различные элементы интерфейса (кнопки, окна редактирования, списки), функционально связанные друг с другом. Такая функциональная связь должна поддерживаться и зрительной связью — объединением соответствующих элементов в рамках одной панели. Панели **Panel** могут также использоваться для организации инструментальных панелей, полос состояния и т.п., хотя для этих целей имеются и специализированные компоненты, которые будут рассмотрены позднее.

Одним из назначений панелей является также группирование таких управляющих элементов, как **RadioButton** — радиокнопки (см. раздел 5.4). Все радиокнопки, расположенные на панели, работают как согласованная группа: в любой момент может быть выбрана только одна из них. Аналогично согласованной группой работают и расположенные на панели быстрые кнопки **SpeedButton** (см. раздел 5.3), если они имеют одинаковое значение свойства **GroupIndex**. В то же время **SpeedButton**, расположенные на разных панелях или на панели и форме, не образуют связанной группы даже при одинаковом значении **GroupIndex**.

Внешний вид панели **Panel** определяется совокупностью параметров **BevelInner** — стиль внутренней части панели, **BevelOuter** — стиль внешней части панели, **BevelWidth** — ширина внешней части панели, **BorderStyle** — стиль бордюра, **BorderWidth** — ширина бордюра. Результат сочетания некоторых значений этих параметров показан на рис. 7.1. Верхняя панель соответствует значениям параметров по умолчанию. Нижняя панель соответствует случаю, когда не определен стиль ни одной из областей панели (значения всех параметров равны **None**). В этом случае сама панель никак не выделяется на форме. Видна только надпись на ней (свойство **Caption**), если надпись задана, и, конечно, видны те компоненты, которые размещаются на панели.

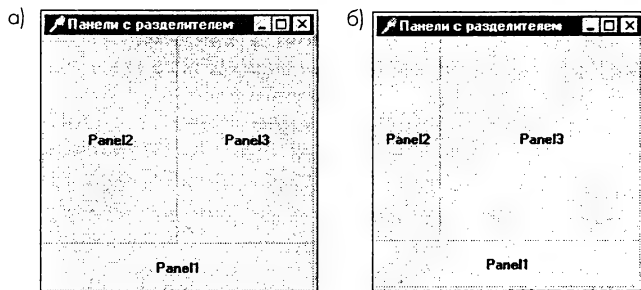
Если вы строите приложение, в котором разрешаете пользователю изменять размер окна, надо позаботиться о том, чтобы синхронно с этим изменялись и размеры панелей. В разделе 3.3.2 уже коротко говорилось о свойствах **Aline**, **Anchor** и **Constraints**, которые позволяют решить эту задачу. Но в ряде приложений этого мало. В зависимости от ситуации какие-то панели в приложении могут оказаться временно перегруженными отображаемыми данными, а другие в значительной степени пустыми. В этих случаях полезно предоставить пользователю возможность перемещать границы, разделяющие различные панели, изменяя их относительные размеры. Пример такой возможности вы можете увидеть в программе Windows «Проводник».

В библиотеке Delphi имеется специальный компонент — **Splitter**, который позволяет легко осуществить это. Рассмотрим это на примере. Пусть вы хотите иметь в приложении форму, содержащую три панели, которые располагаются так, как показано на рис. 7.2. При изменении пользователем размеров окна панель **Panel1** должна в любом случае занимать всю нижнюю часть окна, не увеличиваясь в высоту. Панель **Panel2** должна занимать левую часть окна, изменяя при изменении размеров окна свою высоту, но не изменяя ширину. А панель **Panel3** должна занимать всю остывшуюся часть окна. Кроме того мы хотим обеспечить пользователю возможность изменять положение границы между панелями **Panel2** и **Panel3**, расширяя одну из этих панелей и соответственно сжимая другую.

Исходя из наших требований свойство **Align** надо установить у панели **Panel1** в **alBottom** (выравнивание по нижнему краю окна), у панели **Panel2** — в **alLeft** (выравнивание по левому краю окна), у панели **Panel3** — в **alClient** (выравнивание по всей оставшейся части клиентской области окна). А между **Panel2** и **Panel3** надо разместить **Splitter** — разделитель.

Рис. 7.2

Панели с разделителем



Последовательность проектирования такой формы может быть следующей:

1. Разместите на форме панель **Panel1** и задайте у нее **Align = alBottom**. Панель займет нижнюю часть окна.
2. Разместите на форме панель **Panel2** и задайте у нее **Align = alLeft**. Панель займет левую часть окна.
3. Разместите на форме разделитель **Splitter** и задайте у него **Align = alLeft** (впрочем, это значение **Align** установлено по умолчанию). Разделитель прижмется к правой стороне панели **Panel2**, которая уже выровнена в ту же сторону. Разделитель всегда надо выравнивать только после выравнивания соответствующей панели, так как иначе он прижмется просто к краю формы.
4. После этого можно разместить на форме панель **Panel3** и задать у нее **Align = alClient**. Разделитель окажется зажатым между **Panel2** и **Panel3**.

Откомпилируйте приложение, запустите его на выполнение и убедитесь (рис. 7.2), что, потянув курсором мыши за границу раздела между панелями, вы можете перемещать эту границу.

Закройте приложение и посмотрите в Инспекторе объектов свойства компонента **Splitter**. Свойство **ResizeStyle** определяет поведение разделителя при перемещении его пользователем. По умолчанию это свойство равно **rsPattern**. Это означает (как вы могли видеть в эксперименте), что пока пользователь тянет курсором мыши границу, сам разделитель не перемещается и панели тоже остаются прежних размеров. Перемещается только шаблон линии, указывая место намечаемого перемещения границы. Лишь после того, как пользователь отпустит кнопку мыши, разделитель переместится и панели изменят свои размеры. Практически такая же картина наблюдается, если установить **ResizeStyle = rsLine**. При **ResizeStyle = rsUpdate** в процессе перетаски-

вания границы пользователем разделитель тоже перемещается и размеры панелей все время меняются. Это, может быть, удобно, если пользователь хочет установить размер панели таким, чтобы на ней была видна какая-то конкретная область. Но так как процесс перетаскивания в этом случае сопровождается постоянной перерисовкой панелей, наблюдается неприятное мерцание изображения. Так что этот режим можно рекомендовать только в очень редких случаях. Если установить **ResizeStyle = rsNone**, то в процессе перетаскивания границы не перемещаются ни сама граница, ни изображающая ее линия. Вряд ли это удобно пользователю, так что я не могу представить случая, когда было бы выгодно использовать этот режим.

Свойство **MinSize** компонента **Splitter** устанавливает минимальный размер в пикселях обеих панелей, между которыми зажат разделитель. Задание такого минимального размера необходимо, чтобы при перемещениях границы панель не сжалась бы до нулевого размера или до такой величины, при которой на ней исчезли бы какие-то необходимые для работы элементы управления. К сожалению, как вы сами можете убедиться, экспериментируя с компонентом **Splitter**, свойство **MinSize** не всегда срабатывает верно. К тому же это свойство относится к обоим панелям, граница между которыми перемещается, а в ряде случаев желательно отдельно установить различные минимальные размеры одной и другой панели. Это проще (и надежнее) сделать, задав в панелях соответствующие значения свойства **Constraints**. Например, если вы в описанном выше тестовом приложении зададите в свойстве **Constraints** в панели **Panel2** значение **MinWidth = 50**, а в панели **Panel3** **MinWidth = 100**, то именно только до таких размеров в процессе выполнения приложения пользователь сможет изменять эти панели.

Компонент **Splitter** имеет событие **OnMoved**, которое наступает после конца перемещения границы. В обработчике этого события можно предусмотреть, если необходимо, упорядочение размещения компонентов на панелях, размеры которых изменились: переместить какие-то метки, изменить размеры компонентов и т.д.

Панель **GroupBox** не имеет таких широких возможностей задания различных стилей оформления, как **Panel**. Но она имеет встроенную рамку с надписью (см. рис. 7.1), которая обычно используется для выделения на форме группы функционально объединенных компонентов. Никаких особых свойств, отличных от уже рассмотренных, панель **GroupBox** не имеет.

Компонент **Bevel** формально не является панелью, он не может служить контейнером для компонентов. Например, с помощью **Bevel** нельзя сгруппировать радиокнопки. Однако, чисто зрительно компонент **Bevel** может использоваться как подобие панели. На рис. 7.1 в правой нижней части окна представлены различные варианты оформления **Bevel**.

Стиль отображения **Bevel** определяется свойством **Style**, которое может принимать значения **bsLowered** — утопленный, и **bsRaised** — приподнятый. А контур компонента определяется свойством **Shape**, которое может принимать значения: **bsBox** — прямоугольник, **bsFrame** — рамка, **bsSpacer** — пунктирная рамка, **bsTopLine**, **bsBottomLine**, **bsLeftLine**, **bsRightLine** — соответственной верхняя, нижняя, левая и правая линии. В зависимости от значения **Style** линии могут быть утопленными или выступающими. Все перечисленные варианты приведены на рис. 7.1.

Остановимся теперь на компоненте **ScrollBar** — панели с прокруткой. Этот компонент предназначен для создания области, в которой могут размещаться компоненты, занимающие площадь большую, чем сам **ScrollBar**. Например, компонент **ScrollBar** можно использовать для размещения длинных текстовых строк или больших инструментальных панелей, которые исходя из соображений экономии площади окна нецелесообразно отображать целиком. В примере рис. 7.1 в **ScrollBar** помещена панель с надписью: «Это **ScrollBar**, содержащая панель с длинной надписью». В пределах **ScrollBar** видна только часть этой панели. Если размеры **ScrollBar** меньше, чем размещенные компоненты, то появляются полосы прокрутки, которые позволяют пользователю перемещаться по всем размещенным в **ScrollBar** компонентам.

Разместить в пределах небольшой области **ScrollBar** большие компоненты или много компонентов, занимающих в сумме большую площадь, можно в процессе проектирования, например, с помощью такого приема. Увеличьте временно размер **ScrollBar** так, чтобы в этом компоненте поместилось все, что вы хотите разместить. Проведите необходимое размещение. А затем сократите размеры **ScrollBar** до требуемых.

Свойство **BorderStyle** определяет стиль рамки компонента **ScrollBar**. Свойство **AutoScroll** позволяет задать автоматическое появление необходимых полос прокрутки, если размер размещенных компонентов превышает размер области по горизонтали, вертикали или в обоих измерениях. Если по каким-то соображениям это нежелательно, вы можете сами управлять появлением горизонтальной и вертикальной полос с помощью свойств **Horz-**

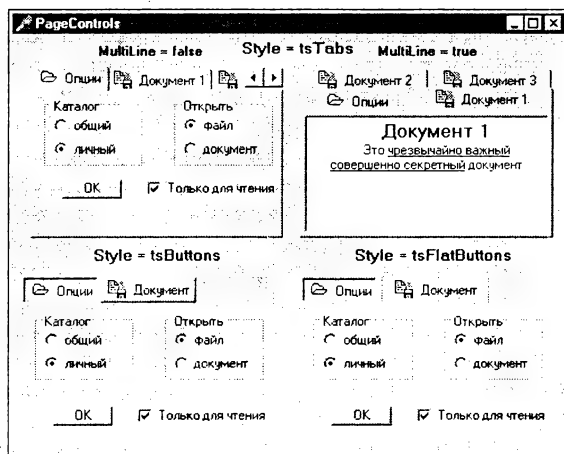
ScrollBar и **VertScrollBar** соответственно. Но в этом случае вы должны сами задавать ряд свойств полосы прокрутки и, прежде всего, **Range** — размер в пикселях прокручиваемой области. Значение перемещения при однократном нажатии пользователем кнопки прокрутки может рассчитываться компонентом автоматически исходя из размеров области и окна, если свойство полосы прокрутки **Smooth** установлено в **true**. В противном случае вы должны задать величину единичного перемещения в свойстве **Increment**.

7.3 Многостраничные панели — компоненты **TabControl**, **PageControl**, **TabSet**, **TabbedNoteBook**, **NoteBook**

Многостраничные панели позволяют экономить пространство окна приложения, размещая на одном и том же месте страницы разного содержания. На рис. 7.3 показаны различные формы отображения многостраничного компонента **PageControl**. Начнем рассмотрение многостраничных панелей именно с этого компонента.

Рис. 7.3

Иллюстрация различных вариантов панели **PageControl**



Перенесите компонент **PageControl** на форму. Чтобы задавать и редактировать страницы этого компонента, надо щелкнуть на нем правой кнопкой мыши. Во всплывшем меню вы можете видеть команды: **New Page** — создать новую страницу, **Next Page** — переключиться на следующую страницу, **Previous Page** — переключиться на предыдущую страницу.

Каждая создаваемая вами страница является объектом типа **TTabSheet**. Это панель, на которой можно размещать любые управляющие компоненты, окна редактирования и т.п. После того, как вы создадите несколько страниц, выделите одну из них, щелкнув в ее середине, и посмотрите ее свойства в Инспекторе Объектов. Страница имеет следующие основные свойства:

Name	Имя, по которому можно ссылаться на страницу
Caption	Надпись, которая появляется на ярлычке закладки
PageIndex	Индекс страницы, по которому можно ссылаться на страницу
ImageIndex	Индекс изображения, которое может появляться на ярлычке закладки

Из общих свойств компонента **PageControl** можно отметить:

Style	Определяет стиль отображения компонента: tsTabs — закладки (верхние компоненты на рис. 7.3), tsButtons — кнопки (левый нижний компонент на рис. 7.3), tsFlatButtons — плоские кнопки (правый нижний компонент на рис. 7.3)
MultiLine	Определяет, будут ли закладки размещаться в несколько рядов, если все они не помещаются в один ряд (на рис. 7.3 сверху два одинаковых компонента, но в левом MultiLine = false , а в правом — true ; примером компонента с MultiLine = false является также знакомая вам палитра компонентов в Delphi)
TabPosition	Определяет место расположения ярлычков закладок: tpBottom — внизу, tpLeft — слева, tpRight — справа и tpTop — сверху компонента (это значение по умолчанию и именно оно задано в примерах рис. 7.3)
TabHeight и TabWidth	Высота и ширина ярлычков закладок в пикселях. Если значения этих параметров заданы равными 0, то размеры ярлычков определяются автоматически по размерам надписей на них
Images	Ссылка на компонент ImageList (см. раздел 9.3), который содержит список изображений на ярлычках. Свойства ImageIndex страниц содержат индексы, соответствующие именно этому списку
ScrollOpposite	Определяет способ перемещения закладок при размещении их в несколько рядов (опробуйте экспериментально, как это свойство влияет на поведение компонента)

ActivePage	Имя активной страницы
Pages [Index:Integer]	Доступ к странице по индексу (первая страница имеет индекс 0) . Свойство только для чтения
PageCount	Количество страниц. Свойство только для чтения

В компоненте имеется ряд методов, позволяющих оперировать страницами, создавать их, уничтожать, переключать. Посмотрите их во встроенной справке Delphi. Основные события компонента — **OnChanging** и **OnChange**. Первое из них происходит непосредственно перед переключением на другую страницу после щелчка пользователя на новой закладке. При этом в обработчик события передается по ссылке параметр **AllowChange** — разрешение переключения. Если в обработчике задать **AllowChange = false**, то переключение не произойдет. Событие **OnChange** происходит сразу после переключения.

Рассмотрим теперь компонент **TabControl**. Внешне этот компонент выглядит так же, как **PageControl**, и имеет много тех же свойств: **Style**, **MultiLine**, **TabPosition**, **TabHeight**, **TabWidth**, **Images**, **ScrollOpposite**, тех же событий: **OnChanging** и **OnChange**. Но принципиальное отличие его от **PageControl** заключается в том, что **TabControl** не имеет множества панелей (страниц). Компонент представляет собой одну страницу с управляющим элементом типа кнопки со многими положениями. И надо написать соответствующие обработчики событий **OnChanging** и **OnChange**, чтобы определить, что именно должно происходить на панели при переключениях закладок пользователем. У компонента имеется еще одно свойство — **MultySelect**, позволяющее множественный выбор закладок. Если это свойство установлено в **true**, то в обработчиках событий надо описать реакцию на такой выбор пользователя.

Число закладок и их надписи определяются свойством **Tabs** типа **TStrings**. В нем вы можете задать надписи закладок. Сколько строчек надписей вы укажете, столько будет закладок. Текущее состояние переключателя определяется свойством **TabIndex**. Вы можете установить его в процессе проектирования, чтобы определить исходное состояние переключателя. А затем в обработчиках событий **OnChanging** и **OnChange** можете читать это свойство, чтобы определить, что именно выбрал пользователь.

Применять компонент **TabControl** имеет смысл в тех приложениях, в которых нужен многопозиционный переключатель. Вы можете, конечно, имитировать с помощью **TabControl** поведение, аналогичное компоненту **PageControl**. Для этого доста-

точно, например, расположить в пределах **TabControl** две закрывающие друг друга панели и в обработчик события **OnChange** вставить оператор:

```
if TabControll1.TabIndex=0  
  then Panel2.Visible:=false  
  else Panel2.Visible:=true;
```

Если **Panel2** — верхняя панель, то при выборе первой закладки (**TabIndex** = 0) она будет делаться невидимой и под ней будет проступать нижняя панель.

Но подобная имитация **PageControl** не имеет смысла, так как проще использовать сам компонент **PageControl**. А **TabControl** надо применять, если требуются какие-то перестроения в рамках одной панели.

Теперь коротко остановимся на компонентах **TabSet**, **TabbedNoteBook** и **NoteBook**. Эти компоненты применяются в Delphi 1 и не рекомендуются для применения в 32-разрядных приложениях.

Компонент **TabbedNoteBook** является аналогом многостраничной панели **PageControl**. Только многие одинаковые у этих панелей свойства называются по-разному. Основное свойство — **Pages**, определяющее число страниц и надписи закладок. Свойство **ActivePage** определяет надпись активной страницы. Свойство **PageIndex** определяет индекс активной страницы (0 — первая страница). Так что узнать, какая страница активна, можно или по значению **ActivePage**, или по **PageIndex**.

В обработчик события **OnChange**, происходящего при переключении пользователем страницы, передается параметр **NewTab**, равный индексу новой страницы, и **AllowChange** — разрешение переключения. Для запрета переключения можно в обработчике задать **AllowChange** = **false**.

Рассмотренный компонент **TabbedNoteBook** является как бы соединением двух компонентов: пачки панелей (страниц) **NoteBook** и набора закладок **TabSet**. Эти два компонента могут использоваться и раздельно. Компонент **TabSet** во многом аналогичен рассмотренному ранее 32-разрядному компоненту **TabControl**. Это многопозиционный управляющий элемент, который сам по себе не имеет никакой панели. Его основное свойство — **Tabs** типа **TStrings**. Задавая строки этого свойства вы тем самым определяете число закладок и их надписи. Свойства **StartMargin** и **EndMargin** определяют поля — расстояния крайних закладок от краев компонента. Сами закладки всегда направлены вниз. Поэтому компонент **TabSet** надо располагать внизу управляемого им компонента. Свойство **AutoScroll** определяет появление

кнопок при большом количестве закладок, которые позволяют пользователю прокручивать полосу закладок, как это делается в компонентах **PageControl** и **TabControl** при **MultiLine = false**. Индекс выбранной закладки определяется свойством **TabIndex**, значение которого можно устанавливать и можно читать в обработчике события **OnChange**, происходящего при смене пользователем закладки и идентичного такому событию в компоненте **TabbedNoteBook**.

Компонент **NoteBook** является пачкой панелей, имена и количество которых определяются свойством **Pages**, как в компоненте **TabbedNoteBook**. Индекс выбранной страницы определяется свойством **PageIndex**. В этом компоненте отсутствует управляющий элемент — закладки. Так что страницы можно переключать какими-то кнопками, переключать их в зависимости от действий пользователя, в зависимости от отображаемых данных и т.п. Компоненты **NoteBook** и **TabSet** могут быть, конечно, объединены программно в компонент, аналогичный **TabbedNoteBook**. Для этого достаточно в обработчик события **OnChange** компонента **TabSet** вставить оператор

```
Notebook1.PageIndex:=NewTab;
```

Но подобное использование этих компонентов вряд ли целесообразно: уж лучше использовать непосредственно **TabbedNoteBook**.

7.4 Инструментальные панели — компоненты **ToolBar** и **PageScroller**

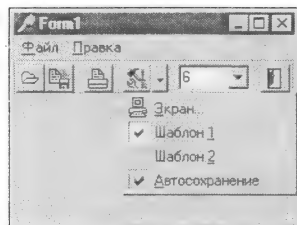
Как уже говорилось выше, инструментальные панели можно создавать и не прибегая к специальным компонентам. Можно поместить на форму простейшую панель **Panel**, разместить на ней быстрые кнопки **SpeedButton** и панель готова. Остается только написать для кнопок соответствующий код. Но специализированные компоненты, которые мы рассмотрим в этом разделе, дают, конечно, дополнительные возможности для построения инструментальных панелей.

Начнем рассмотрение компонентов, которые используются для построения различных инструментальных панелей, с компонента **ToolBar**. Пример панели, построенной на основе этого компонента, приведен на рис. 7.4.

Если вы поместите компонент **ToolBar** на форму, то по умолчанию он расположится вверху, поскольку его свойство **Align** по умолчанию равно **alTop**. Если вы хотите, чтобы панель распола-

Рис. 7.4

Пример инструментальной панели ToolBar



галась иначе, установите **Align = alNone**, после чего можете придать панели любую форму и расположить ее в любом месте.

Занесение компонентов на панель **ToolBar** можно, в принципе, осуществлять обычным способом — переносом их из палитры компонентов. Но для занесения кнопок имеется и более простой вариант. Щелкните на **ToolBar** правой кнопкой мыши и выберите из всплывшего меню команду **New Button**. На форме появится очередная кнопка — объект типа **TToolButton**. Это не совсем обычная кнопка, так как в дальнейшем вы увидите, что внешне она может не походить на кнопку. Ее вид и поведение определяется ее свойством **Style**, которое по умолчанию равно **tbsButton** — кнопка. Другие возможные стили мы рассмотрим позднее. А как кнопка этот объект очень похож на кнопку **SpeedButton**. Только изображение на кнопке определяется не свойством **Glyph**, а свойством **ImageIndex**. Оно определяет индекс изображения, хранящегося во внешнем компоненте **ImageList** (см. раздел 9.3). Указание на этот компонент может задаваться такими свойствами компонента **ToolBar**, как **Images**, **DisabledImages** (указывает на список изображений кнопок в недоступном состоянии) и **HotImages** (указывает на список изображений кнопок в моменты, когда над ними перемещается курсор мыши).

Свойство **MenuItem** позволяет задать раздел главного или контекстного меню (см. разделы 6.1 и 6.2), который дублируется данной кнопкой. При установке этого свойства, если в соответствующем разделе меню было задано изображение и установлен текст подсказок (свойство **Hint**), то это же изображение появится на кнопке и тот же текст появится в свойстве **Hint** кнопки. Передадутся из раздела меню в кнопку также значения свойств **Enabled** (доступность) и **Visible** (видимость). Правда, все это передастся в кнопку только в момент установки свойства **MenuItem**. Если в процессе дальнейшего проектирования вы измените соответствующие свойства раздела меню, это не отразится на свойствах кнопки. Но если вы сотрете значение **MenuItem**, а потом установите его снова, то в кнопке зафиксируются новые значения свойств раздела меню.

Свойство **Wrap**, установленное в **true**, приводит к тому, что после этой кнопки ряд кнопок на панели прерывается и следующие кнопки размещаются в следующем ряду.

Теперь вернемся к свойству **Style**, задающему стиль кнопки. Значение **Style = tbsCheck** определяет, что после щелчка пользователя на кнопке она остается в нажатом состоянии. Повторный щелчок на кнопке возвращает ее в отжатое состояние. Поведение такой кнопки подобно кнопкам **SpeedButton** и определяется аналогичными свойствами **AllowAllUp** и **Down** (см. раздел 5.3). Если при этом в нескольких кнопках установлено свойство **Grouped = true**, то эти кнопки образуют группу, из которой только одна кнопка может находиться в нажатом состоянии.

Значение **Style = tbsDropDown** соответствует кнопке в виде выпадающего списка. Этот стиль удобен для воспроизведения выпадающего меню. Если для подобной кнопки задать в качестве свойства **MenuItem** головной раздел меню, то в выпадающем списке автоматически будут появляться разделы выпадающего меню. В примере рис. 7.4 стиль **tbsDropDown** имеет четвертая слева кнопка. В ней в качестве свойства **MenuItem** задан раздел Опции из меню, рассмотренного в разделе 6.1 и представленного на рис. 6.2. При **Style = tbsDropDown** можно вместо свойства **MenuItem** задать свойство **DropDownMenu**, определяющее контекстное меню (компонент **PopupMenu**), которое будет отображаться в выпадающем списке.

Значение **Style = tbsSeparator** приводит к появлению разделителя, позволяющего отделить друг от друга кнопки разных функциональных групп. Значение **Style = tbsDivider** приводит к появлению разделителя другого типа — в виде вертикальной линии. Разделитель можно ввести и из контекстного меню **ToolBar**, выбрав команду **New Separator**.

Свойство кнопки **Indeterminate** задает ее третье состояние — не нажатая и не отпущенная. Это свойство можно устанавливать в **true** во время выполнения, если в данном режиме кнопка не доступна.

Свойство **Marked** выделяет кнопку.

Мы рассмотрели занесение на панель кнопок. Но в инструментальных панелях нередко используются и выпадающие списки. Например, для задания размера шрифта. Не представляет труда перенести на панель **ToolBar** такие компоненты, как **ComboBox** (это изображено на рис. 7.4), **SpinEdit** и др.

Из общих свойств компонента **ToolBar** следует еще отметить **ButtonHeight** и **ButtonWidth** — высота и ширина кнопок в пикселях, и **Wrapable** — автоматический перенос кнопок в следую-

щий ряд панели, если они не помещаются в предыдущем. Такой перенос осуществляется и во время проектирования, и во время выполнения при изменении пользователем размеров панели.

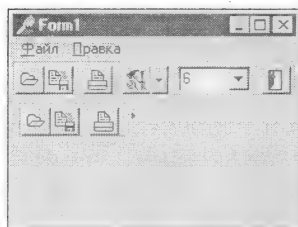
Свойства, определяющие вид панели **ToolBar: BorderWidth** — ширина бордюра, **EdgeInner** и **EdgeOuter** — стиль изображения внутренней и внешней части панели (утопленный или выступающий), **EdgeBorders** — определяет изображение отдельных сторон панели (левой, правой, верхней, нижней).

Мы рассмотрели построение инструментальной панели на основе компонента **ToolBar**. Но полоса может быть очень длинной и не помещаться в отведенном ей месте формы. Примером является палитра компонентов Delphi. В этих случаях может помочь компонент **PageScroller**, обеспечивающий прокрутку панели. Собственно говоря, **PageScroller** может прокручивать любой компонент, не обязательно панель **ToolBar**. Например, он может прокручивать какую-то панель вместе с размещенными на ней компонентами. В этом отношении он напоминает рассмотренный в разделе 7.2 компонент **ScrollBar**. Но есть и заметные различия между этими двумя компонентами: **PageScroller** прокручивает только один компонент и только в одном направлении — горизонтальном или вертикальном. Да и оформление управления прокруткой у **PageScroller** не похоже на полосы прокрутки в **ScrollBar**.

Пример применения компонента **PageScroller** показан на рис. 7.5. Это тот же пример, что и на рис. 7.4. В верхней части окна показана та же инструментальная панель, что и на рис. 7.4. А ниже показана идентичная панель, но заключенная в небольшое окно **PageScroller** и снабженная кнопкой прокрутки.

Рис. 7.5

Пример инструментальной панели и ее прокрутки компонентом **PageScroller**



Основное свойство компонента **PageScroller** — **Control**. Оно указывает компонент, который должен размещаться и прокручиваться в окне **PageScroller**. Благодаря наличию этого свойства вы можете проектировать свою инструментальную панель, например, **ToolBar**, не помещая ее заранее в окно **PageScroller** и не задумываясь о ее размере. А после того, как вы спроектировали

панель, можно ввести на форму компонент **PageScroller** и установить его свойство **Control**. В этот момент ваша инструментальная панель переместится в окно компонента **PageScroller** и появится, если необходимо, кнопка прокрутки.

Свойство **Margin** компонента **PageScroller** определяет размер полей в пикселях, которые оставляются между краем окна **PageScroller** и прокручиваемым компонентом. По умолчанию эти поля равны нулю, но надо задать свойству **Margin** некоторое положительное значение. Иначе края прокручиваемого компонента могут быть плохо видны.

Свойство **AutoScroll** определяет, должна ли прокрутка осуществляться автоматически, как только курсор мыши пройдет над кнопкой прокрутки. Опробуйте режим автоматической прокрутки экспериментально. На мой взгляд лучше оставлять значение **AutoScroll** равным **false**, поскольку такая автоматическая прокрутка не очень удобна пользователю.

7.5 Перестраиваемые панели — компоненты **CoolBar** и **ControlBar**

Перестраиваемые панели являются дальнейшим развитием инструментальных панелей. Только в перестраиваемых панелях сами инструментальные панели обычно являются компонентами более сложных образований. Примером перестраиваемой панели может служить панель ИСП Delphi 5, включающая в себя ряд более мелких панелей быстрых кнопок и палитру компонентов. Пользователь может настраивать их, изменять местоположение панелей и т. п.

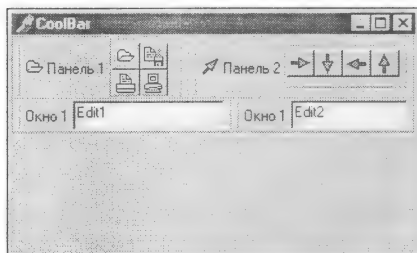
Начнем рассмотрение с компонента **CoolBar**. Он позволяет строить перестраиваемые панели, состоящие из полос (bands). В полосы могут включаться инструментальные панели **ToolBar** и любые другие оконные компоненты: окна редактирования, панели и т. п. Каждый из этих компонентов автоматически снабжается средствами перемещения его пользователем в пределах окна **CoolBar**. В полосы могут вставляться и не оконные компоненты, например, метки. Но они не будут перемещаемыми.

Опробуйте в работе этот компонент. Поместите его на форму. Перенесите на него другие компоненты, например, **ToolBar** и **Edit**. Когда вы размещаете на **CoolBar** очередной компонент, ему отводится отдельная полоса и он растягивается на всю ширину **CoolBar**. Около каждого компонента появляется слева полоска, за которую компонент можно перемещать. Например, взявшись за эту полоску вы можете переместить полосу вместе с ее компо-

нентом в тот ряд, где уже имеется другой компонент. Тогда они расположатся в ряд один левее другого (см. пример на рис. 7.6).

Рис. 7.6

Пример перестраиваемой панели на основе компонента CoolBar



Свойства полос вы можете задавать редактором полос. С этим инструментом вы уже имели дело в разделе 3.6.2, но тогда он фигурировал как редактор заголовков. Вызвать редактор полос можно тремя способами: из Инспектора Объектов кнопкой с многоточием около свойства **Bands**, двойным щелчком на компоненте **CoolBar** или из контекстного меню, выбрав команду **Bands Editor**. В окне этого редактора вы можете перемещаться по полосам, добавлять новые полосы или уничтожать существующие. При перемещении по полосам в окне Инспектора Объектов вы будете видеть свойства полос. Свойство **Control** определяет размещенный на полосе компонент. Свойство **Break** определяет, занимает ли полоса весь соответствующий размер контейнера **CoolBar**, или обрывается. Если вы расположите полосы так, как показано на рис. 7.6, то в левых полосах автоматически установится **Break = true**, а в правых — **Break = false**.

Свойство **Text** задает текст, который может появиться в начале соответствующей полосы. Это свойство можно оставлять пустым. А можно и задать его — надписи «Панель 1», «Панель 2», «Окно 1», «Окно 2» на рис. 7.6 заданы именно этим свойством.

Вместо свойства **Text** (или наряду с ним) можно задать свойство **ImageIndex** — индекс списка изображений **ImageList** (см. раздел 9.3), ссылка на который задается свойством **Images**. Указанные таким образом изображения появятся в начале соответствующих полос (см. верхние полосы на рис. 7.6).

Свойства **MinHeight** и **MinWidth** определяют минимальную высоту и ширину полосы при перестроениях пользователем полос панели.

Свойство **FixedSize** определяет, фиксирован ли размер данной полосы или он может изменяться пользователем. По умолчанию для всех полос **FixedSize = false**, т.е. все полосы перестраивают-

ся. Но при желании размеры некоторых полос можно зафиксировать, задав для них **FixedSize = true**.

Для компонента **CoolBar** в целом, помимо обычных для других панелей свойств, надо обратить внимание на свойство **BandMaximize**. Оно определяет действие, которым пользователь может установить максимальный размер полосы, не перетаскивая ее границу: **bmNone** — такое действие не предусмотрено, **bmClick** — щелчком мыши, **bmDbClick** — двойным щелчком. Наиболее целесообразно, по-видимому, задавать значения **bmDbClick** или **bmNone**, поскольку значение **bmClick** приводит к резкому перестроению полос даже при случайном щелчке мыши.

Свойство **FixedOrder**, если его установить в **true**, не разрешит пользователю в процессе перемещений полос изменять их последовательность. Вероятно, такое задание лучше, чем значение по умолчанию, равное **false**, поскольку чрезмерная свобода для пользователя способна его запутать.

Свойство **Vertical** указывает вертикальное или горизонтальное расположение полос. По умолчанию **Vertical = false**, что соответствует горизонтальным полосам.

Запустите свое тестовое приложение, если вы его построили, и опробуйте его в работе. Вы увидите, как легко пользователь может перестраивать панель.

Еще большую свободу перестроений дает пользователю компонент **ControlBar**, который появился в Delphi 4. Только оформление панели несколько отличается от **CoolBar** и кроме того в ней может широко применяться техника перетаскивания и встраивания **Drag&Doc**, широко используемая в Delphi (начиная с Delphi 4), в частности, используемая в инструментальных панелях среды проектирования Delphi.

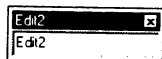
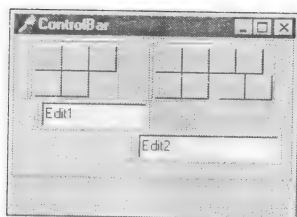
Поместите на форму компонент **ControlBar** и перенесите на него несколько компонентов, например, инструментальных панелей **ToolBar** и окон редактирования **Edit**. Вы увидите (см. рис. 7.7), что каждый компонент, попадая на **ControlBar**, получает полосу захвата, свойственную технологии **Drag&Doc**.

Установите у компонентов, размещенных на **ControlBar**, свойство **DragMode = dmAutomatic** и **DragKind = dkDock**. Это означает автоматическое выполнение операций **Drag&Doc**.

Свойства компонента **ControlBar** **RowSize** и **RowSnap** определяют процедуру встраивания. Свойство **RowSize** задает размеры полос, в которые могут встраиваться компоненты, а **RowSnap** определяет захват полосами встраиваемых компонентов. Свойство **AutoDrag** определяет, можно (при значении **true**), или нельзя

Рис. 7.7

Перестраиваемая панель на основе компонента **ControlBar** (а) и вынесенное из нее окно редактирования (б)



простым перетаскиванием вынести полосу за пределы **ControlBar**.

Запустите приложение и посмотрите на практике широчайшие возможности перестроения панелей. А если вы установили свойство **AutoDrag** в **true**, то вы можете даже вынимать из панели отдельные компоненты и они становятся самостоятельными окнами (см. рис. 7.7 б). Опробуйте в эксперименте различные значения свойства компонента **ControlBar**, и они станут вам более понятны.

7.6 Полоса состояния **StatusBar**

Компонент **StatusBar** представляет собой ряд панелей, отображающих полосу состояния в стиле Windows. Обычно эта полоса размещается внизу формы.

Свойство **SimplePanel** определяет, включает ли полоса состояния одну или множество панелей. Если **SimplePanel = true**, то вся полоса состояния представляет собой единственную панель, текст которой задается свойством **SimpleText**. Если же **SimplePanel = false**, то полоса состояния является набором панелей, задаваемых свойством **Panels**. В этом случае свойство **SizeGrip** определяет, может ли пользователь изменять размеры панелей в процессе выполнения приложения.

Каждая панель полосы состояния является объектом типа **TStatusPanels**. Свойства панелей вы можете задавать специальным редактором наборов. С этим инструментом вы уже имели дело в разделах 3.6.2 и 7.5 при редактировании заголовков и полос. Вызвать редактор можно тремя способами: из Инспектора Объектов кнопкой с многоточием около свойства **Panels**, двойным щелчком на компоненте **StatusBar** или из контекстного меню, выбрав команду **Panels Editor**. В окне редактора вы можете перемещаться по панелям, добавлять новые или уничтожать существующие. При перемещении по панелям в окне Инспектора Объектов вы будете видеть их свойства.

Основное свойство каждой панели — **Text**, в который заносится отображаемый в панели текст. Его можно занести в процессе проектирования, а затем можно изменять программно во время выполнения. Другое существенное свойство панели — **Width** (ширина).

Программный доступ к текстам отдельных панелей можно осуществлять двумя способами: через индексированное свойство **Panels** или через его индексированное подсвойство **Items**. Например, два следующих оператора дадут идентичный результат:

```
StatusBar1.Panels[0].Text := 'текст 1';
```

или

```
StatusBar1.Panels.Items[0].Text := 'текст 1';
```

Оба они напечатают текст «текст 1» в первой панели.

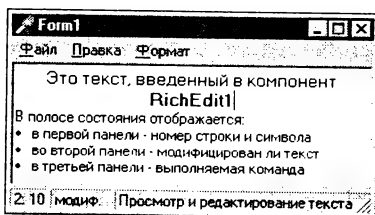
Количество панелей полосы состояния можно определить из подсвойства **Count** свойства **Panels**. Например, следующий оператор очищает тексты всех панелей:

```
for i:= 0 to StatusBar1.Panels.Count -1 do  
  StatusBar1.Panels[i].Text := '';
```

На рис. 7.8 приведен пример текстового редактора на основе компонента **RichEdit**, содержащий полосу состояния. В ее первой панели отображается номер строки и символа, перед которым находится курсор, во второй — отображается, модифицирован текст в окне, или нет. В третьей панели отображается подсказка о назначении компонента, над которым в данный момент расположен курсор мыши.

Рис. 7.8

Пример редактора **RichEdit** с полосой состояния



Для реализации такой полосы состояния надо в обработчиках событий **OnKeyDown**, **OnKeyUp**, **OnMouseDown** и **OnMouseUp** компонента **RichEdit1** и события **OnResize** формы обеспечить выполнение операторов:

```
StatusBar1.Panels[0].Text:=IntToStr(  
  RichEdit1.CaretPos.Y+1) +  
  ':'+IntToStr(RichEdit1.CaretPos.X+1);  
if RichEdit1.Modified  
  then StatusBar1.Panels[1].Text := 'модиф.'  
  else StatusBar1.Panels[1].Text := '';
```

Эти операторы заполняют первые две панели полосы состояния. Занесение в строку состояния подсказок в целом выходит за рамки данной книги. Но, поскольку, говоря о полосе состояния, было бы неправильно обойти этот вопрос, ниже без особых пояснений описывается, как это можно сделать.

В Delphi 5 это делается очень просто с помощью компонента **ApplicationEvents**. Этот компонент помещается на форму и в обработчик его события **OnHint** (см. подробнее в разделе 9.4) заносится оператор, который для приведенного на рис. 7.8 примера имеет вид:

```
StatusBar1.Panels[2].Text := Application.Hint;
```

Он отображает в панели свойство приложения **Application.Hint**, а в это свойство автоматически переносится вторая часть текста подсказки (свойство **Hint** — о нем подробнее сказано ниже) того компонента, над которым перемещается курсор мыши.

В предыдущих версиях Delphi это оформляется более громоздко. Аналогичный оператор надо занести в обработчик события **OnHint** переменной **Application** (см. раздел 9.4). Этот обработчик вам надо создать вручную. Пусть вы назвали его **DisplayHint**.

Прототип этой функции **DisplayHint** можно внести в описание класса формы. Кроме того надо указать приложению на эту функцию как на обработчик события **OnHint**. Это можно сделать, например, задав в обработчике события **OnCreate** формы оператор:

```
Application.OnHint := DisplayHint;
```

В итоге текст вашего модуля может иметь вид (приводится с сокращениями):

```
interface
...
type
  TForm1 = class(TForm)
    StatusBar1: TStatusBar;
    ...
    procedure FormCreate(Sender: TObject);
    procedure DisplayHint(Sender: TObject);
    ...
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```
procedure TForm1.DisplayHint(Sender: TObject);
begin
  StatusBar1.Panels[2].Text := Application.Hint;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnHint := DisplayHint;
end;
...
```

Как видим, все это существенно более громоздко, чем указанные выше возможности, реализованные в Delphi 5.

Чтобы все это работало, надо в свойствах **Hint** тех компонентов, пояснения которых вы хотите отображать в панели состояния, написать соответствующие тексты. Если вы к тому же хотите, чтобы у компонента появлялся ярлычок с короткими подсказками при задержке пользователем мыши над этим компонентом, то текст в свойстве **Hint** должен состоять из двух частей, разделенных символом вертикальной черты «|». Например, в свойстве **Hint** быстрой кнопки, соответствующей разделу меню Сохранить можно ввести текст: «Сохранить|Сохранение документа в файле». Первая часть этого текста будет появляться в ярлычке кнопки, если ее свойство **ShowHint** (показать подсказку) установлено в **true**. А вторую часть описанные выше операторы будут отображать в строке состояния при перемещении курсора мыши над кнопкой, независимо от значения ее свойства **ShowHint**.

7.7 Фреймы

В Delphi 5 введен новый компонент, который помогает поддерживать стилистическое единство приложения. Это **Frame** — фрейм. Он представляет собой нечто среднее между панелью и формой. С формой его роднит то, что он:

- проектируется отдельно, как самостоятельное окно
- имеет свой модуль — файл **.pas**
- имеет возможности наследования, причем даже более широкие, чем у формы, так как может наследоваться даже внутри одного приложения
- может включаться в Депозитарий и использоваться так же, как и форма, включая наследование

С панелью фрейм роднит то, что он:

- не является самостоятельным окном Windows и может отображаться только на форме или другом контейнере

■ имеет свойства, методы, события, подобные панели, а не форме

Таким образом, фрейм — это панель, т.е. некий фрагмент окна приложения, но способный переноситься на разные формы, в разные приложения и допускающий использование преимуществ наследования.

Начать проектирование нового фрейма можно командой File | New Frame или командой File | New и выбором пиктограммы Frame на странице New окна Депозитария. В обоих случаях перед вами откроется окно фрейма, подобное окну формы, а в Редакторе Кода вы увидите текст заготовки модуля фрейма:

```
unit Unit2;

interface // Открытый интерфейс фрейма

{Список подключаемых модулей}
uses Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs;

{Объявление класса фрейма}
type
  TFrame2 = class(TFrame)
  {Сюда Delphi помещает объявления компонентов,
  размещаемых на фрейме. Не добавляйте сюда ничего
  вручную}
  private // Закрытый раздел класса
    { Private declarations }
  {Сюда могут помещаться объявления переменных, функций и
  процедур, включаемых в класс фрейма, но не доступных для
  других модулей}
  public // Открытый раздел класса
    { Public declarations }
  {Сюда могут помещаться объявления переменных, функций и
  процедур, включаемых в класс фрейма и доступных для
  других модулей}
  end;

{Сюда могут помещаться объявления типов, констант,
переменных, функций и процедур, к которым будет доступ
из других модулей, но которые не включаются в класс
фрейма. Они будут едины для всех объектов фреймов}

implementation // Реализация модуля

{$R *.DFM}
{Сюда могут помещаться предложения uses, объявления
типов, констант, переменных, к которым не будет доступа
из других модулей. Они будут едины для всех объектов
```

фреймов. Тут же должны быть реализации всех объявленных в разделе `interface` функций и процедур, а также могут быть реализации любых дополнительных, не объявленных ранее функций и процедур}

`end.`

Комментарии в приведенном тексте поясняют, куда и что можно помещать в модуль. Те переменные, объявления которых вы поместите в объявление класса, будут индивидуальны для каждого объекта фрейма. Объявления имеют обычный для класса вид. Например:

```
A:integer;
```

Переменные, объявления которых вы поместите вне объявления класса, будут едины для всех объектов фрейма. Они объявляются как обычные переменные. Например:

```
var A:integer;
```

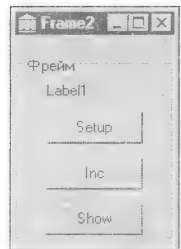
На фрейм вы можете так же, как на форму, переносить и размещать любые компоненты, устанавливать их свойства, писать обработчики их событий и т.п.

Давайте создадим чисто тестовый фрейм, чтобы на его примере продемонстрировать проектирование фрейма, его использование, доступ к различным его элементам и наследование свойств.

Начните новое приложение и выполните команду `File | New Frame`. Перенесите на фрейм групповую панель **GroupBox** (см. раздел 7.2). Перенесите на панель метку **Label** и три кнопки **Button**. Разместите все эти компоненты примерно так, как показано на рис. 7.9, изменив соответственно их надписи (**Caption**) и назвав кнопки соответственно **BSetup**, **BInc**, **BShow**.

Рис. 7.9

Пример фрейма



Давайте введем в наш модуль в разных местах объявления целых переменных, а в обработчики событий кнопок введем операторы, манипулирующие ими и отображающие результат в метке. Модуль фрейма может приобрести следующий вид:

```
unit UFrame;

interface

uses Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs, StdCtrls;

type
  TFrame2 = class(TFrame)
    GroupBox1: TGroupBox;
    BSetup: TButton;
    Label1: TLabel;
    BInc: TButton;
    BShow: TButton;
    procedure BSetupClick(Sender: TObject);
    procedure BIncClick(Sender: TObject);
    procedure BShowClick(Sender: TObject);
  private
    { Private declarations }
  {Переменная A видна только в данном модуле}
    A:integer;
  public
    { Public declarations }
  {Переменная B видна в других модулях через объект фрейма}
    B:integer;
  end;
  {Переменная C видна в других модулях}
  var C:integer;

implementation

{$R *.DFM}
{Переменная D видна только в данном модуле}
var D:integer;

procedure TFrame2.BSetupClick(Sender: TObject);
begin
  A:=1;
  B:=1;
  C:=1;
  D:=1;
  Label1.Caption := 'A='+IntToStr(A)+ ' B='+IntToStr(B)+
    ' C='+IntToStr(C)+ ' D='+IntToStr(D);
end;

procedure TFrame2.BIncClick(Sender: TObject);
begin
  Inc(A);
  Inc(B);
  Inc(C);
```

```

Inc(D);
Labell1.Caption := 'A='+IntToStr(A) + ' B='+IntToStr(B) +
                  ' C='+IntToStr(C) + ' D='+IntToStr(D);
end;

procedure TFrame2.BShowClick(Sender: TObject);
begin
  Labell1.Caption := 'A='+IntToStr(A) + ' B='+IntToStr(B) +
                    ' C='+IntToStr(C) + ' D='+IntToStr(D);
end;

end.

```

В модуле введены переменные:

- **A** — введена в закрытый раздел класса; видна только в процедурах данного класса в этом модуле; независимые друг от друга переменные **A** будут содержаться в каждом объекте фрейма
- **B** — введена в открытый раздел класса; в других модулях можно получить доступ к **B** через имя объекта фрейма; независимые друг от друга переменные **B** будут содержаться в каждом объекте фрейма
- **C** — введена в открытый интерфейс вне описания класса; доступна в других модулях; имеется единственный экземпляр **C**, независимо от числа объектов фреймов
- **D** — введена в реализацию класса; доступна только в данном модуле; имеется единственный экземпляр **D**, независимо от числа объектов фреймов

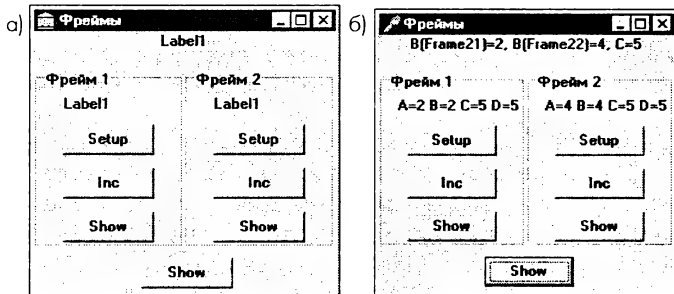
Введенные в модуль обработчики щелчков на кнопках обеспечивают сброс всех переменных на 1 (процедура **TFrame2.BSetupClick**), увеличение всех переменных на 1 (процедура **TFrame2.BIncClick**), отображение текущего состояния переменных (процедура **TFrame2.BShowClick**).

Теперь давайте разместим несколько экземпляров фрейма на форме. Перейдите в основную форму приложения и выберите в палитре компонентов **Frame** (первая кнопка на странице **Standard**). Появится диалоговое окно, в котором будет спрашиваться, какой фрейм вы хотите разместить на форме. Выберите ваш фрейм **Frame2** и он появится на форме. Можете отбуксировать его, как обычный компонент, в нужное место. Повторите эту операцию еще раз и разместите на форме второй фрейм (рис. 7.10). Добавьте кнопку и метку, задав ее свойство **Align** равным **alTop** и свойство **Alignment** равным **taCenter**.

Вы получили форму, содержащую два объекта — фрейма. Можете изменить какие-то свойства объектов. Например, изменить надписи (**Caption**) групповых панелей **GroupBox** (см. рис. 7.10 а).

Рис. 7.10

Пример использования фреймов: форма (а) и приложение в работе (б)



После того, как вы изменили эти свойства, они перестают наследоваться из класса фрейма. А остальные свойства продолжают наследоваться. В этом легко убедиться. Перейдите в модуль фрейма (рис. 7.9) и измените у фрейма стиль шрифта (**Font.Style**) на жирный. Вы увидите, что в обоих объектах главной формы шрифт тоже станет жирным. Верните во фрейме шрифт на обычный и он синхронно изменится в объектах. А теперь установите в одном из фреймов на форме шрифт жирным. Повторив после этого эксперимент с изменением шрифта в исходном фрейме, вы увидите, что теперь шрифт меняется только в том объекте формы, в котором вы его не изменяли вручную. Таким образом объекты наследуют только те свойства, которые не были в них установлены вручную.

Теперь давайте напишем обработчик щелчка на кнопке главной формы. Прежде всего взгляните на текст модуля этой формы. Вы увидите, что в нем в описании класса формы появились две строки:

```
Frame21: TFrame2;  
Frame22: TFrame2;
```

Это объявления объектов фреймов. Все компоненты, размещенные на фреймах, напрямую из модуля формы не видны. Доступ к ним можно получить только через объекты **Frame21** и **Frame22**. Имена компонентов, размещенных во фреймах, локальные. Несмотря на то, что во фреймах имеются кнопки с именами **BShow**, вы можете назвать тем же именем кнопку на форме.

Поместите в обработчик щелчка на этой кнопке оператор

```
Label1.Caption := 'B(Frame21)='+IntToStr(Frame21.B) +  
                ', B(Frame22)='+IntToStr(Frame22.B) +  
                ', C='+IntToStr(C);
```

Он отображает в метке **Label1** значения переменных **B** объектов фреймов и значение переменной **C** класса фрейма. Значения

переменных **A** и **D** отобразить невозможно, поскольку эти переменные недоступны из внешних модулей. Если вы попытаетесь отобразить их значения, компилятор выдаст сообщение об ошибке.

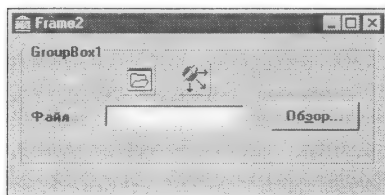
Сохраните ваше приложение, оттранслируйте его и выполните. Манипулируя кнопками вы легко сможете убедиться (см. рис. 7.10 б), что переменные **A** и **B** независимы для каждого фрейма, а переменные **C** и **D** одинаковы. Точнее оба фрейма оперируют с одними и теми же переменными **C** и **D**.

Рассмотренный фрейм не имел никакого практического значения. Давайте построим более полезный пример. Во многих диалогах при установке различных опций фигурирует фрагмент, фрейм которого показан на рис. 7.11. Фрагмент включает в себя панель **GroupBox**, окно редактирования, в котором пользователь может написать имя файла, и кнопку Обзор, которая позволяет выбрать файл в стандартном диалоге Windows открытия файла. Если путь к файлу длинный, то полное имя файла с путем может не помещаться в окне редактирования. Поэтому полезно для него предусмотреть всплывающее окно, которое отображало бы полное имя файла вместе с путем и всплывало бы, если пользователь задержал над ним курсор мыши.

Давайте построим подобный фрейм и опробуем его в работе. Начните новое приложение и выполните команду File | New Frame. Перенесите на фрейм групповую панель **GroupBox**. Перенесите в эту панель окно редактирования **Edit**, кнопку **Button**, диалог **OpenDialog** (см. раздел 8.2) и компонент **ApplicationEvents** — перехватчик событий приложения (см. раздел 9.4). Расположите компоненты примерно так, как показано на рис. 7.11.

Рис. 7.11

Фрейм выбора файла



Задайте в свойстве **Filter** диалога **OpenDialog** какой-то фильтр файлов, например, «все файлы|*.*». Свойство **ShowHint** (показать ярлычок подсказки) в компонентах **Edit** и **Button** установите в **true**. В кнопке **Button** кроме того можете написать текст подсказки **Hint**, например, «Выбор файла|Выбор файла из каталога».

В обработчик события **OnShowHint** компонента **ApplicationEvents** занесите оператор:

```

if HintInfo.HintControl = Edit1
then begin
    HintStr:=Edit1.Text;
    ApplicationEvents1.CancelDispatch;
end;

```

Этот оператор в момент, когда должен отображаться ярлычок, проверяет, не является ли источником этого события (**HintInfo.HintControl**) окно редактирования **Edit1**. Если да, то текст ярлычка (**HintStr**) подменяется текстом, содержащимся в окне редактирования и принимаются меры (метод **CancelDispatch**), чтобы это событие не обрабатывалось другими компонентами **ApplicationEvents**, которые могут присутствовать в приложении. Пояснение всех этих операций см. в разделе 9.4.

Теперь введите в модуль фрейма глобальную переменную **FileName** типа **string**, в которой будет отображаться выбранный файл, В обработчик щелчка на кнопке введите оператор

```

if OpenFileDialog1.Execute
then begin
    Edit1.Text := OpenFileDialog1.FileName;
    FileName := OpenFileDialog1.FileName;
end;

```

который вызывает диалог открытия файла и помещает в окно редактирования **Edit1** и в переменную **FileName** имя файла, выбранного пользователем, вместе с путем к нему.

В обработчик события **OnExit** компонента **Edit1** поместите оператор

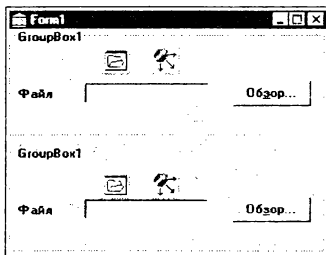
```
FileName:=Edit1.Text;
```

заносящий в переменную **FileName** имя файла, если пользователь не пользовался диалогом, а просто написал в окне имя файла.

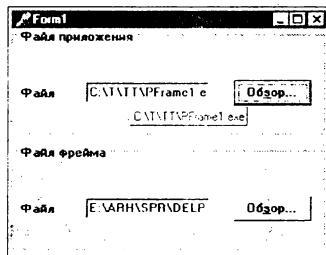
Программирование фрейма закончено. Теперь создайте тестовую программу, использующую этот фрейм. Предположим, что вам нужно разместить на форме два фрагмента, описанных вами во фрейме. Перейдите в основной модуль вашего приложения и

Рис. 7.12 а)

Приложение с двумя фреймами выбора файла: его форма (а) и приложение во время выполнения (б)



б)



разместите на форме так, как вы уже делали, два объекта вашего фрейма (рис. 7.12 а).

Теперь вы можете поменять что-то в размещенных на форме объектах фреймов, изменить надписи групповых панелей, шрифты и т.п. Сохраните ваше приложение вместе с модулем фрейма, оттранслируйте его и проверьте в работе (рис. 7.12 б).

Вы разработали достаточно полезный фрейм и хотели бы его сохранить для использования в будущих приложениях. Это легко сделать, внося его в Депозиторий. Щелкните на своем фрейме правой кнопкой мыши и выберите из всплывшего меню раздел Add To Repository. Перед вами откроется окно, представленное на рис. 7.13. В верхнем его окне редактирования Title вы должны написать название вашего фрейма — подпись под его пиктограммой при входе в Депозиторий. В следующем окне — Description можете написать более развернутое пояснение. Его может увидеть пользователь, войдя в Депозиторий, щелкнув правой кнопкой мыши и выбрав во всплывшем меню форму отображения View Details. В выпадающем списке Page вы можете выбрать страницу Депозитария, на которой хотите разместить пиктограмму своего фрейма. Впрочем, вы можете указать и новую страницу с новым заголовком (Мои формы на рис. 7.13). В результате она появится в Депозитарии.

В окне Author вы можете указать сведения о себе как об авторе. Наконец, если стандартная пиктограмма вас не устраивает, вы можете выбрать другую, щелкнув на кнопке Browse... После выполнения всех этих процедур щелкните на кнопке ОК и ваш фрейм окажется включенным в Депозиторий.

Теперь вы можете использовать его в последующих ваших приложениях. Для этого вам надо будет выполнить команду File | New и в открывшемся диалоговом окне New Items отыскать ваш фрейм (рис. 7.14).

Рис. 7.13

Окно добавления
фрейма в Депозиторий

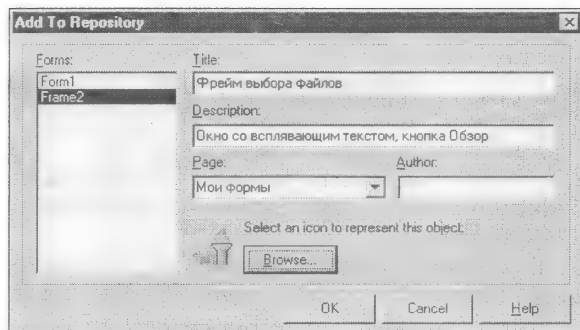
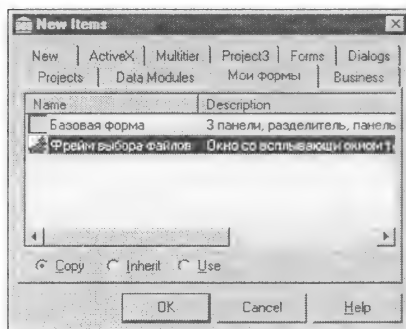


Рис. 7.1

Окно New Items с включенным новым фреймом



В нижней части окна расположены три радиокнопки, которые определяют, как именно вы хотите заимствовать фрейм из Депозитария: Copy — копировать, Inherit — наследовать, Use — использовать. Если включена кнопка Copy, то файлы фрейма просто будут скопированы в ваше приложение. При этом никакой дальнейшей связи между исходным фреймом и копией не будет. Вы можете спокойно изменять свойства вашей копии и это никак не отразится на фрейме, хранящемся в Депозитарии. А если вы в дальнейшем что-то измените во фрейме, хранящемся в Депозитарии, то эти изменения никак не затронут вашего приложения, куда вы до этого скопировали фрейм.

При включенной кнопке Inherit вы получите в своем проекте фрейм, наследующий размещенному в Депозитарии. Это значит, что если вы что-то измените во фрейме, хранящемся в Депозитарии, то это отразится при перекомпиляции во всех проектах, которые наследуют этот фрейм. Однако, изменения в наследуемых фреймах никак не скажутся на свойствах фрейма, хранящегося в Депозитарии.

При включенной кнопке Use вы получите режим использования. В этом случае в ваш проект включится сам фрейм, хранящийся в Депозитарии. Значит любое изменение свойств фрейма, сделанное в вашем проекте, отразится и на хранящемся в Депозитарии фрейме, и на всех проектах, наследующих или использующих этот фрейм (при их перекомпиляции).

Таким образом, режим Inherit целесообразно использовать для всех модулей вашего проекта, а режим Use — для изменения базового фрейма. Тогда усовершенствование вами базового фрейма будет синхронно сказываться на всех модулях проекта при их перекомпиляции.

Введенный в Delphi 5 компонент фрейм благодаря использованию наследования позволяет обеспечить единство стилистиче-

ских решений не только внутри приложения, но и в рамках серии разрабатываемых вами приложений. Вам достаточно один раз разработать какие-то часто применяемые фреймы, включить их в Депозитарий, а затем вы можете использовать их многократно во всех своих проектах.

8.1 Общая характеристика компонентов — диалогов

В приложениях часто приходится выполнять стандартные действия: открывать и сохранять файлы, задавать атрибуты шрифтов, выбирать цвета палитры, производить контекстный поиск и замену и т.п.

Разработчики Delphi позаботились о том, чтобы включить в библиотеку простые для использования компоненты, реализующие соответствующие диалоговые окна. Они размещены на странице Dialogs. В таблице 8.1 приведен перечень этих диалогов.

Таблица 8.1. Системные диалоги и их фрагменты

Пикто- грамма	Компонент	Страница	Описание
	OpenDialog «Открыть файл»	Dialogs	Предназначен для создания окна диалога «Открыть файл».
	SaveDialog «Сохранить файл»	Dialogs	Предназначен для создания окна диалога «Сохранить файл как».
	OpenPictureDialog «Открыть рисунок»	Dialogs	Предназначен для создания окна диалога «Открыть рисунок», открывающего графический файл. Начиная с Delphi 3.
	SavePictureDialog «Сохранить рисунок»	Dialogs	Предназначен для создания окна диалога «Сохранить рисунок» — сохранение изображения в графическом файле. Начиная с Delphi 3.
	FontDialog «Шрифты»	Dialogs	Предназначен для создания окна диалога «Шрифты» — выбор атрибутов шрифта.
	ColorDialog «Цвет»	Dialogs	Предназначен для создания окна диалога «Цвет» — выбор цвета.

Пиктограмма	Компонент	Страница	Описание
	PrintDialog «Печать»	Dialogs	Предназначен для создания окна диалога «Печать».
	PrinterSetupDialog «Установка принтера»	Dialogs	Предназначен для создания окна диалога «Установка принтера».
	FindDialog «Найти»	Dialogs	Предназначен для создания окна диалога «Найти» — контекстный поиск в тексте.
	ReplaceDialog «Заменить»	Dialogs	Предназначен для создания окна диалога «Заменить» — контекстная замена фрагментов текста.
	FileListBox (список файлов)	Win 3.1	Отображает список всех файлов каталога.
	DirectoryListBox (структура каталогов)	Win 3.1	Отображает структуру каталогов диска.
	DriveComboBox (список дисков)	Win 3.1	Выпадающий список доступных дисков.
	FilterComboBox (список фильтров)	Win 3.1	Выпадающий список фильтров для поиска файлов.
	DirectoryOutline (дерево каталогов)	Samples	Пример компонента, используемого для отображения структуры каталогов выбранного диска.

Последние четыре компонента в таблице 8.1 являются не законченными диалогами, а их фрагментами, позволяющими строить свои собственные диалоговые окна.

Все диалоги являются невидимыми компонентами, так что место их размещения на форме не имеет значения. При обращении к этим компонентам вызываются стандартные диалоги, вид которых зависит от версии Windows и настройки системы. Так что при запуске одного и того же приложения на компьютерах с разными системами диалоги будут выглядеть по-разному. Например, при русифицированной версии Windows все их надписи будут русскими, а при англоязычной версии надписи будут на английском языке.

Основной метод, которым производится обращение к любому диалогу, — **Execute**. Эта функция открывает диалоговое окно и, если пользователь произвел в нем какой-то выбор, то функция возвращает **true**. При этом в свойствах компонента — диалога запоминается выбор пользователя, который можно прочесть и использовать в дальнейших операциях. Если же пользователь в диалоге нажал кнопку Отмена или клавишу Esc, то функция **Execute** возвращает **false**. Поэтому стандартное обращение к диалогу имеет вид:

```
if <имя компонента — диалога> Execute  
then <операторы, использующие выбор пользователя>;
```

8.2 Диалоги открытия и сохранения файлов — компоненты **OpenDialog**, **SaveDialog**, **OpenPictureDialog**, **SavePictureDialog**

Компоненты **OpenDialog** — диалог «Открыть файл» и **SaveDialog** — диалог «Сохранить файл как...», пожалуй, используются чаще всего, в большинстве приложений. Примеры открываемых ими диалоговых окон приведены на рис. 8.1 и 8.2.

Рис. 8.1

Диалоговое окно открытия файла

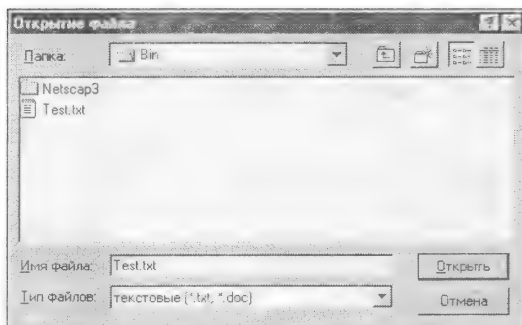
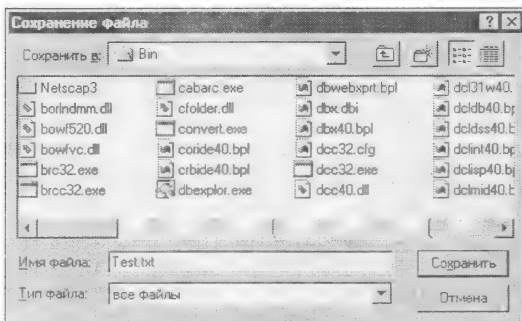


Рис. 8.2

Диалоговое окно сохранения файла

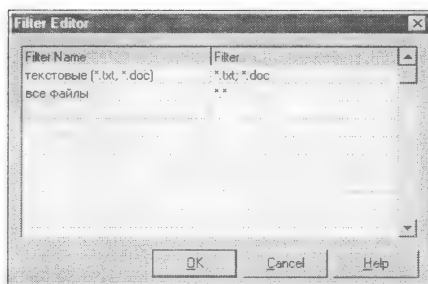


Все свойства этих компонентов одинаковы, только их смысл несколько различен для открытия и закрытия файлов. Основное свойство, в котором возвращается в виде строки выбранный пользователем файл, — **FileName**. Значение этого свойства можно задать и перед обращением к диалогу. Тогда оно появится в диалоге как значение по умолчанию в окне Имя файла (см. рис. 8.1, 8.2).

Типы искоемых файлов, появляющиеся в диалоге в выпадающем списке Тип файла (рис. 8.1, 8.2), задаются свойством **Filter**. В процессе проектирования это свойство проще всего задать с помощью редактора фильтров, который вызывается нажатием кнопки с многоточием около имени этого свойства в Инспекторе Объектов. При этом открывается окно редактора, вид которого представлен на рис. 8.3. В его левой панели Filter Name вы записываете тот текст, который увидит пользователь в выпадающем списке Тип файла диалога. А в правой панели Filter записываются разделенные точками с запятой шаблоны фильтра. В примере рис. 8.3 задано два фильтра: текстовых файлов с расширениями **.txt** и **.doc** и любых файлов с шаблоном ***.***.

Рис. 8.3

Окно редактора фильтров



После выхода из окна редактирования фильтров заданные вами шаблоны появятся в свойстве **Filter** в виде строки вида:

```
текстовые (*.txt, *.doc)|*.*; *.doc|все файлы|*.*
```

В этой строке тексты и шаблоны разделяются вертикальными линиями. В аналогичном виде, если требуется, можно задавать свойство **Filter** программно во время выполнения приложения.

Свойство **FilterIndex** определяет номер фильтра, который будет по умолчанию показан пользователю в момент открытия диалога. Например, значение **FilterIndex** = 1 задает по умолчанию первый фильтр.

Свойство **InitialDir** определяет начальный каталог, который будет открыт в момент начала работы пользователя с диалогом. Если значение этого свойства не задано, то открывается текущий

каталог или тот, который был открыт при последнем обращении пользователя к соответствующему диалогу в процессе выполнения данного приложения.

Свойство **DefaultExt** определяет значение расширения файла по умолчанию. Если значение этого свойства не задано, пользователь должен указать в диалоге полное имя файла с расширением. Если же задать значение **DefaultExt**, то пользователь может писать в диалоге имя без расширения. В этом случае будет принято заданное расширение.

Свойство **Title** позволяет вам задать заголовок диалогового окна. Если это свойство не задано, окно открывается с заголовком, определенным в системе (например, «Открытие файла» в окне на рис. 8.1). Но вы можете задать и свой заголовок, подсказывающий пользователю ожидаемые действия. Например, «Укажите имя открываемого файла».

Свойство **Options** определяет условия выбора файла. Множество опций, которые вы можете установить программно или во время проектирования, включает:

ofAllowMultiSelect	Позволяет пользователю выбирать несколько файлов
ofCreatePrompt	В случае, если пользователь написал имя несуществующего файла, появляется замечание и запрос, надо ли создать файл с заданным именем
ofEnableIncludeNotify	Разрешает посылать в диалог сообщения
ofEnableSizing	Разрешает пользователю изменять размер диалогового окна
ofExtensionDifferent	Этот флаг, который можно прочесть после выполнения диалога, показывает, что расширение файла, выбранного пользователем, отличается от DefaultExt
ofFileMustExist	В случае, если пользователь написал имя несуществующего файла, появляется сообщение об ошибке
ofHideReadOnly	Удаляет из диалога индикатор Открыть только для чтения
ofNoChangeDir	После щелчка пользователя на кнопке ОК восстанавливает текущий каталог, независимо от того, какой каталог был открыт при поиске файла

ofNoDereferenceLinks	Запрещает переназначать клавиши быстрого доступа в диалоговом окне
ofNoLongNames	Отображаются только не более 8 символов имени и трех символов расширения
ofNoNetworkButton	Убирает из диалогового окна кнопку поиска в сети. Действует только если флаг ofOldStyleDialog включен
ofNoReadOnlyReturn	Если пользователь выбрал файл только для чтения, то генерируется сообщение об ошибке
ofNoTestFileCreate	Запрещает выбор в сети защищенных файлов и не доступных дисков при сохранении файла
ofNoValidate	Не позволяет писать в именах файлов неразрешенные символы, но не мешает выбирать файлы с неразрешенными символами
ofOldStyleDialog	Создает диалог выбора файла в старом стиле (см. рис. 8.4)
ofOverwritePrompt	В случае, если при сохранении файла пользователь написал имя существующего файла, появляется замечание, что файл с таким именем существует, и запрашивается желание пользователя переписать существующий файл
ofPathMustExist	Генерирует сообщение об ошибке, если пользователь указал в имени файла несуществующий каталог
ofReadOnly	По умолчанию устанавливает индикатор Открыть только для чтения при открытии диалога
ofShareAware	Игнорирует ошибки нарушения условий коллективного доступа и разрешает, несмотря на них, производить выбор файла
ofShowHelp	Отображает в диалоговом окне кнопку Справка

По умолчанию все перечисленные опции, кроме **ofHideReadOnly**, выключены. Но, как видно из их описания, многие из них полезно включить перед вызовом диалогов.

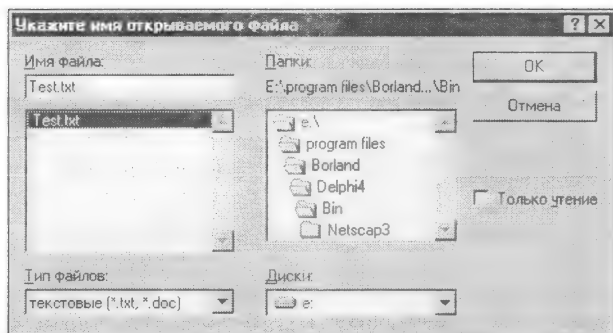
Если вы разрешаете с помощью опции **ofAllowMultiSelect** множественный выбор файлов, то список выбранных файлов можно прочитать в свойстве **Files** типа **TStrings**.

В приведенной таблице даны опции, используемые в 32-разрядных версиях Delphi. В Delphi 1 диалоговое окно имеет вид, представленный на рис. 8.4. Аналогичный вид имеет диалог и в 32-разрядных версиях Delphi при включении опции **ofOldStyleDialog**. В примере рис. 8.4 диалог открыт с заданным значением

свойства **Title** и заданный текст отображается в заголовке окна. Кроме того, в этом примере выключена опция **ofHideReadOnly** (в Delphi 1 она выключена по умолчанию), что привело к появлению индикатора «Только чтение».

Рис. 8.4

Диалог в старом стиле при включенной опции **ofOldStyleDialog** и выключенной опции **ofHideReadOnly**



В компонентах диалогов открытия и сохранения файлов предусмотрена возможность обработки ряда событий. Такая обработка может потребоваться, если рассмотренных опций, несмотря на их количество, не хватает, чтобы установить все диктуемые конкретным приложением ограничения на выбор файлов. Событие **OnCanClose** возникает при нормальном закрытии пользователем диалогового окна после выбора файла. При отказе пользователя от диалога — нажатии кнопки **Отмена**, клавиши **Esc** и т.д. событие **OnCanClose** не наступает. В обработке события **OnCanClose** вы можете произвести дополнительные проверки выбранного пользователем файла и, если по условиям вашей задачи этот выбор недопустим, вы можете известить об этом пользователя и задать значение **false** передаваемому в обработчик параметру **CanClose**. Это не позволит пользователю закрыть диалоговое окно.

Можно также написать обработчики событий **OnFolderChange** — изменение каталога, **OnSelectionChange** — изменение имени файла, **OnTypeChange** — изменение типа файла. В этих обработчиках вы можете предусмотреть какие-то сообщения пользователю.

Теперь приведем примеры использования диалогов **OpenDialog** и **SaveDialog**. Пусть ваше приложение включает окно редактирования **Memor1**, в которое по команде меню **Открыть** вы хотите загружать текстовый файл, а после каких-то изменений, сделанных пользователем, — сохранять по команде **Сохранить текст** в том же файле, а по команде **Сохранить как...** — в файле с другим именем.

Введите на форму компоненты — диалоги **OpenDialog** и **SaveDialog**. Предположим, что вы оставили их имена по умолчанию — **OpenDialog1** и **SaveDialog1**. Поскольку после чтения файла вам надо запомнить его имя, чтобы знать под каким именем потом его сохранять, вы можете определить для этого имени переменную, назвав ее, например, **FName**:

```
var FName:string;
```

Тогда обработка команды Открыть может сводиться к следующему оператору:

```
if OpenDialog1.Execute  
then begin  
    FName:=OpenDialog1.FileName;  
    Mem1.Lines.LoadFromFile(FName);  
end;
```

Этот оператор вызывает диалог, проверяет, выбрал ли пользователь файл (если выбрал, то функция **Execute** возвращает **true**), после чего имя выбранного файла (**OpenDialog1.FileName**) сохраняется в переменной **FName** и файл загружается в текст **Mem1** методом **LoadFromFile**.

Обработка команды Сохранить выполняется оператором

```
Mem1.Lines.SaveToFile(FName);
```

В данном случае нет необходимости обращаться к какому-то диалогу, поскольку имя файла известно: оно хранится в переменной **FName**.

Обработка команды Сохранить как... выполняется операторами:

```
SaveDialog1.FileName:=FName;  
if SaveDialog1.Execute  
then begin  
    FName:=OpenDialog1.FileName;  
    Mem1.Lines.SaveToFile(FName);  
end;
```

Первый из этих операторов присваивает свойству **FileName** компонента **SaveDialog1** запомненное имя файла. Это имя по умолчанию будет предложено пользователю при открытии диалога Сохранить как.... Следующий оператор открывает диалог и, если пользователь выбрал в нем файл, запоминает новое имя файла и сохраняет в файле с этим именем текст компонента **Mem1**.

Мы рассмотрели диалоги открытия и сохранения файлов произвольного типа. Начиная с Delphi 3 в библиотеке имеются специализированные диалоги открытия и закрытия графических файлов: **OpenPictureDialog** и **SavePictureDialog**. Диалоговые окна, открываемые этими файлами, приведены на рис. 8.5 и 8.6. От окон, открываемых компонентами **OpenDialog** и **SaveDialog**

(рис. 8.1, 8.2), они отличаются удобной возможностью просматривать изображения в процессе выбора файла.

Рис. 8.5

Диалоговое окно открытия файла изображения

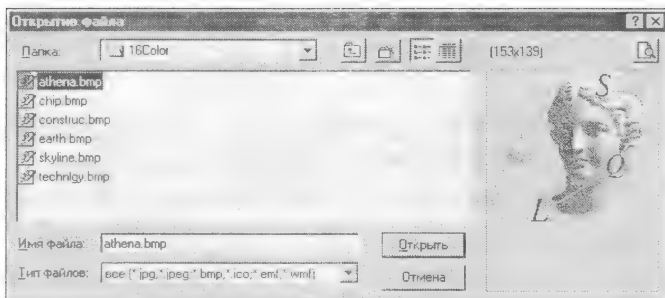
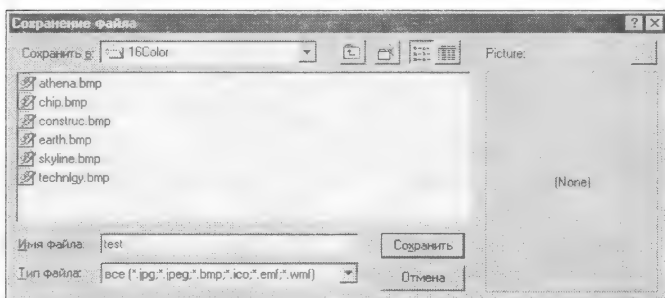


Рис. 8.6

Диалоговое окно сохранения файла изображения



Свойства компонентов **OpenPictureDialog** и **SavePictureDialog** ничем не отличаются от свойств компонентов **OpenDialog** и **SaveDialog**. Единственное отличие — заданное значение по умолчанию свойства **Filter** в **OpenPictureDialog** и **SavePictureDialog**. В этих компонентах заданы следующие фильтры:

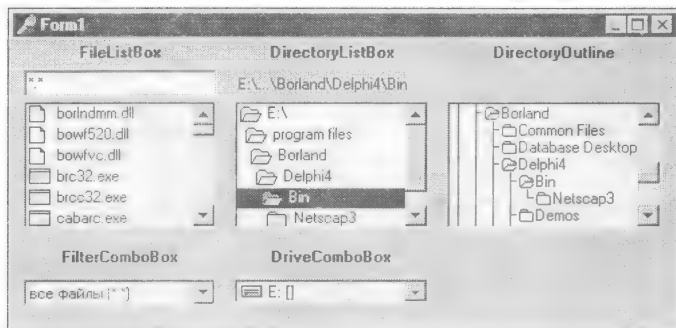
All (*.jpg;*.jpeg;*.bmp;*.ico;*.emf;*.wmf)	*.jpg;*.jpeg;*.bmp;*.ico; *.emf;*.wmf
JPEG Image File (*.jpg)	*.jpg
JPEG Image File (*.jpeg)	*.jpeg
Bitmaps (*.bmp)	*.bmp
Icons (*.ico)	*.ico
Enhanced Metafiles (*.emf)	*.emf
Metafiles (*.wmf)	*.wmf

В этих фильтрах перечислены все типы графических файлов, с которыми может работать диалог. Так что вам остается удалить, если хотите, фильтры тех файлов, с которыми вы не хотите работать, добавить, может быть, фильтр «Все файл (*.*)» и перевести на русский язык названия типов.

8.3 Фрагменты диалогов — компоненты **DriveComboBox**, **DirectoryListBox**, **FilterComboBox**, **FileListBox** и **DirectoryOutline**

Помимо законченных диалогов работы с файлами, в Delphi имеется ряд компонентов, представляющих собой фрагменты диалогов: выпадающие списки дисков (драйверов) — **DriveComboBox** и фильтров (масок) файлов — **FilterComboBox**, списки каталогов — **DirectoryListBox** и файлов — **FileListBox**, дерево каталогов — **DirectoryOutline**. Внешний вид этих компонентов вы можете увидеть на рис. 8.7. Компоненты работы с файловой системой облегчают вам создание собственных диалоговых окон, что нередко требуется. Например, вы можете захотеть включить в диалоговое окно отображение каких-то характеристик файлов (размера, даты создания и т.п.) или оперативный просмотр содержания текстовых файлов. Тогда вам очень пригодятся готовые компоненты работы с файлами. Правда, все они, кроме **DirectoryOutline**, предназначены в основном для Delphi 1 и в последующих версиях Delphi расположены на странице Win 3.1 палитры компонентов. Это значит, что они не рекомендуются для 32-разрядных приложений. Но, во-первых, у вас остается компонент **DirectoryOutline**. Кроме того, никто не запрещает вам все-таки использовать и остальные компоненты в любых приложениях Delphi. И, наконец, если уж вы хотите четко следовать рекомендации не использовать первые четыре фрагмента диалогов в

Рис. 8.7
Компоненты — фрагменты диалогов



32-разрядных приложениях, вы можете разработать свои аналогичные компоненты, используя обычный компонент **ComboBox**.

Начнем рассмотрение компонентов работы с файловой системой с компонента **DriveComboBox** — выпадающего списка дисков (драйверов). При размещении на форме этот компонент автоматически отображает список имеющихся на компьютере дисков. Во время выполнения приложения вы можете прочитать имя выбранного пользователем диска в свойстве **Drive**, а строку, содержащуюся в окне списка — в свойстве **Text**.

Свойство **TextCase** задает регистр отображения: **tcUpperCase** — в верхнем регистре, **tcLowerCase** — в нижнем.

Связать компонент **DriveComboBox** со списком каталогов, отображаемых компонентом **DirectoryListBox**, можно во время проектирования через свойство **DirList** компонента **DriveComboBox**. Это свойство может указывать на компонент типа **DirectoryListBox**. Можно обеспечить связь этих двух типов компонентов и программно, включив в обработчик события **OnChange** компонента **DriveComboBox** оператор

```
DirectoryListBox1.Drive:=DriveComboBox1.Drive;
```

Этот оператор задает имя диска, выбранное пользователем в компоненте **DriveComboBox1**, свойству **Drive** списка каталогов **DirectoryListBox1**.

Аналогичным оператором можно обеспечить связь компонента **DriveComboBox** с деревом каталогов и файлов в компоненте **DirectoryOutline**:

```
DirectoryOutline1.Drive:=DriveComboBox1.Drive;
```

Рассмотрим теперь выпадающий список фильтров — компонент **FilterComboBox**. Его основное свойство — **Filter**, которое задается так же, как в описанных ранее диалогах. К отдельным частям фильтра — тексту и маске, можно получить доступ через свойства **Text** и **Mask** соответственно. Связь компонента со списком файлов типа **TFileListBox** можно установить, задав свойство **FileList**.

Компонент **DirectoryListBox** отображает список каталогов диска, заданного свойством **Drive**. Значение этого свойства можно установить программно во время выполнения. Как уже говорилось выше, связь этого свойства с выбранным пользователем диском в компоненте **DriveComboBox** устанавливается или программно, или с помощью свойства **DirectoryListBox** компонента **DriveComboBox**.

Связь списка каталогов с компонентом типа **TFileListBox**, отображающим список файлов, осуществляется с помощью свойства **FileList**. Можно также использовать результаты выбора по-

льзователем каталога, читая свойство **Directory** в обработчике события **OnChange**.

С компонентом **DirectoryListBox** можно также связать метку типа **Label**. В этой метке будет отображаться путь к текущему каталогу. Если путь не уместается в метке, он автоматически отображается в сокращенном виде (см. рис. 8.7) с помощью функции **MinimizeName**. Метка, отображающая каталог, указывается в свойстве **DirLabel**.

Список файлов содержится в компоненте **FileListBox**. Его свойства **Drive**, **Directory** и **Mask** определяют соответственно диск, каталог и маску файлов. Эти свойства можно устанавливать программно или связывая описанным ранее способом компонент **FileListBox** с компонентами **DriveComboBox**, **DirectoryListBox** и **FilterComboBox**. Свойство **FileType** позволяет включать в список не все файлы, а только те, которые имеют соответствующие атрибуты. Свойство **FileType** представляет собой множество, указывающее типы включаемых файлов. Элементы этого множества могут иметь значения: **ftReadOnly** — только для чтения, **ftHidden** — невидимые, **ftSystem** — системные, **ftVolumeID** — обозначения дисков, **ftDirectory** — каталоги, **ftArchive** — архивные, **ftNormal** — не имеющие особых атрибутов.

Свойство **ShowGlyphs** разрешает или исключает показ пиктограмм файлов (в примере рис. 8.7 это свойство = **true**).

Свойство **MultiSelect** разрешает выбор нескольких файлов.

Основное свойство, в котором можно прочитать имя выбранного пользователем файла — **FileName**.

Со списком файлов может быть связано окно редактирования **Edit**, в котором отображается выбранный файл (см. окно над списком файлов на рис. 8.7). На этот список указывает устанавливаемое во время проектирования свойство **FileEdit**.

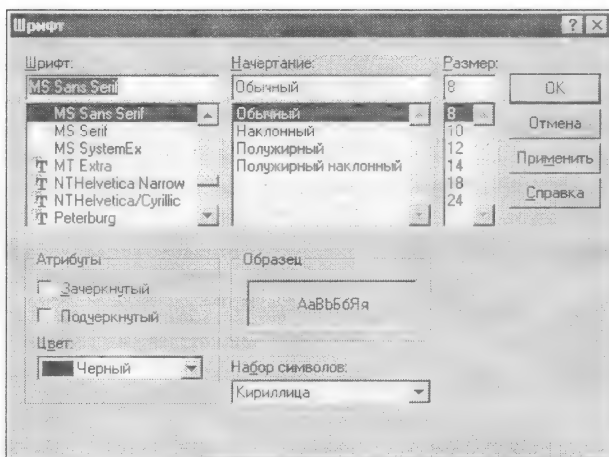
Теперь рассмотрим компонент **DirectoryOutline**, содержащий дерево каталогов. В этом компоненте значение диска устанавливается свойством **Drive**. Текущий каталог, выбранный пользователем, можно прочитать в свойстве **Directory**. Свойство **TextCase** определяет стиль отображения имен каталогов: **tcLowerCase** — преобразование к нижнему регистру, **tcUpperCase** — к верхнему, **tcAsIs** — без преобразования (этот режим использован в примере рис. 8.7). Остальные свойства идентичны компоненту **Outline**, на основе которого построен данный пример. Вы можете найти исходный текст этого примера в каталоге ...\\source\\samples.

8.4 Диалог выбора шрифта — компонент FontDialog

Компонент **FontDialog** вызывает диалоговое окно выбора атрибутов шрифта, представленное на рис. 8.8. В нем пользователь может выбрать имя шрифта, его стиль (начертание), размер и другие атрибуты.

Рис. 8.8

Диалоговое окно выбора атрибутов шрифта



Основное свойство компонента — **Font** типа **TFont** (см. раздел 3.1), в котором вы можете задать при желании начальные установки атрибутов шрифта и в котором вы можете прочесть значения атрибутов, выбранные пользователем в процессе диалога.

Свойства **MaxFontSize** и **MinFontSize** устанавливают ограничения на максимальный и минимальный размеры шрифта. Если значения этих свойств равны 0 (по умолчанию), то никакие ограничения на размер не накладываются. Если же значения свойств заданы (обычно это целесообразно делать исходя из размеров компонента приложения, для которого выбирается шрифт), то в списке **Размер** диалогового окна (см. рис. 8.8) появляются только размеры, укладывающиеся в заданный диапазон. При попытке пользователя задать недопустимый размер ему будет выдано предупреждение вида «Размер должен лежать в интервале ...» и выбор пользователя отменится. Свойства **MaxFontSize** и **MinFontSize** действуют только при включенной опции **fdLimitSize** (см. ниже).

Свойство **Device** определяет, из какого списка возможных шрифтов будет предложен выбор в диалоговом окне: **fdScreen** —

из списка экрана (по умолчанию), **fdPrinter** — из списка принтера, **fdBoth** — из обоих.

Свойство **Options** содержит множество опций:

fdAnsiOnly	Отображать только множество шрифтов символов Windows, не отображать шрифтов со специальными символами
fdApplyButton	Отображать в диалоге кнопку Применить независимо от того, предусмотрен ли обработчик события OnApply
fdEffects	Отображать в диалоге индикаторы специальных эффектов (подчеркивание и др.) и список Цвет
fdFixedPitchOnly	Отображать только шрифты с постоянной шириной символов
fdForceFontExist	Позволять пользователю выбирать шрифты только из списка, запрещать ему вводить другие имена
fdLimitSize	Разрешить использовать свойства MaxFontSize и MinFontSize , ограничивающие размеры шрифта
fdNoFaceSel	Открывать диалоговое окно без предварительно установленного имени шрифта
fdNoOEMFonts	Удалять из списка шрифтов шрифты OEM.
fdScalableOnly	Отображать только масштабируемые шрифты, удалять из списка не масштабируемые (шрифты bitmap)
fdNoSimulations	Отображать только шрифты и их начертания, напрямую поддерживаемые файлами, не отображая шрифты, в которых жирный стиль и курсив синтезируются
fdNoSizeSel	Открывать диалоговое окно без предварительно установленного размера шрифта
fdNoStyleSel	Открывать диалоговое окно без предварительно установленного начертания шрифта
fdNoVectorFonts	Удалять из списка векторные шрифты (типа Roman или Script для Windows 1.0)
fdShowHelp	Отображать в диалоговом окне кнопку Справка
fdTrueTypeOnly	Предлагать в списке только шрифты TrueType
fdWysiwyg	Предлагать в списке только шрифты, доступные и для экрана, и для принтера, удаляя из него аппаратно зависимые шрифты

По умолчанию все эти опции, кроме **fdEffects**, отключены.

Если установить опцию **fdApplyButton**, то при нажатии пользователем кнопки Применить возникает событие **OnApply**, в обработчике которого вы можете написать код, который применит выбранные пользователем атрибуты, не закрывая диалогового окна.

Приведем примеры применения компонента **FontDialog**. Пусть ваше приложение включает окно редактирования **Memo1**, шрифт в котором пользователь может выбирать командой меню Шрифт. Вы ввели в приложение компонент **FontDialog**, имя которого по умолчанию **FontDialog1**. Тогда обработчик команды Шрифт может иметь вид:

```
if FontDialog1.Execute  
then Memo1.Font.Assign(FontDialog1.Font);
```

Приведенный оператор вызывает диалог выбора атрибутов шрифта и, если пользователь произвел выбор, то значения всех выбранных атрибутов, содержащиеся в свойстве **FontDialog1.Font**, присваиваются атрибутам окна редактирования, содержащимся в свойстве **Memo1.Font**. Шрифт в окне редактирования немедленно изменится.

Если вы установите в компоненте **FontDialog1** опцию **fdApplyButton**, то можете написать обработчик события **OnApply**:

```
Memo1.Font.Assign(FontDialog1.Font);
```

Тогда пользователь может наблюдать изменения в окне **Memo1**, нажимая в диалоговом окне кнопку Применить и не прерывая диалога. Это очень удобно, так как позволяет пользователю правильно подобрать атрибуты шрифта.

Если в качестве окна редактирования в вашем приложении вы используете **RichEdit**, то можете предоставить пользователю выбирать атрибуты шрифта для выделенного фрагмента текста или для вновь вводимого текста. Тогда выполнение команды меню Шрифт может осуществляться операторами:

```
if FontDialog1.Execute then  
RichEdit1.SelectAttributes.Assign(FontDialog1.Font);
```

Вы можете разрешить пользователю изменять шрифт не только отдельных компонентов, но и всех компонентов и надписей на форме. Это осуществляется оператором:

```
if FontDialog1.Execute  
then Font.Assign(FontDialog1.Font);
```

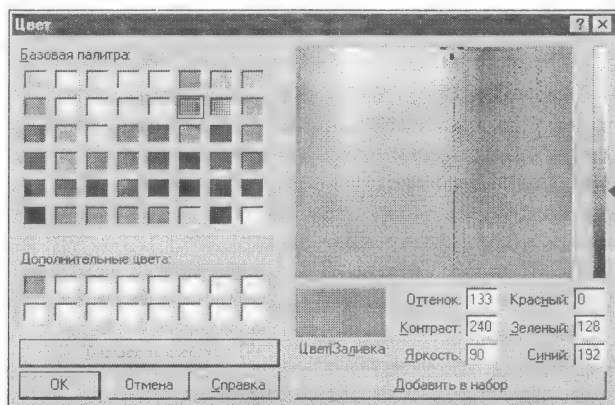
В этом операторе свойство **Font** без ссылки на компонент подразумевает шрифт формы.

8.5 Диалог выбора цвета — компонент ColorDialog

Компонент **ColorDialog** вызывает диалоговое окно выбора цвета, представленное на рис. 8.9. В нем пользователь может выбрать цвет из базовой палитры или, нажав кнопку **Определить цвет**, раскрыть дополнительную панель (на рис. 8.9 она раскрыта), позволяющую синтезировать цвет, отличный от базовых. Синтезированный цвет можно добавить кнопкой **Добавить** в набор в палитру дополнительных цветов на левой панели и использовать его в дальнейшем.

Рис. 8.9

Диалоговое окно выбора цвета



Основное свойство компонента **ColorDialog** — **Color**. Это свойство соответствует тому цвету, который выбрал в диалоге пользователь. Если при вызове диалога желательно установить некоторое начальное приближение цвета, это можно сделать, установив **Color** предварительно во время проектирования или программно.

Свойство **CustomColors** типа **TStrings** позволяет задать заказные цвета дополнительной палитры. Каждый цвет определяется строкой вида

`<Имя цвета>=<шестнадцатеричное представление цвета>;`

Имена цветов задаются от **ColorA** (первый цвет) до **ColorP** (шестнадцатый, последний). Например, строка

`ColorA=808022`

задает первый заказной цвет. При задании цвета 2 младших разряда описывают интенсивность красного цвета, следующие 2 — зеленого, старшие — синего.

Свойство **Options** содержит множество следующих опций:

cdFullOpen	Отображать сразу при открытии диалогового окна панель определения заказных цветов
cdPreventFullOpen	Запретить появление в диалоговом окне кнопки Определить цвет, так что пользователь не сможет определять новые цвета
cdShowHelp	Добавить в диалоговое окно кнопку Справка
cdSolidColor	Указать Windows использовать сплошной цвет, ближайший к выбранному (это обедняет палитру)
cdAnyColor	Разрешать пользователю выбирать любые не сплошные цвета (такие цвета могут быть не равными)

По умолчанию все опции выключены.

Приведем пример применения компонента **ColorDialog**. Если вы хотите, чтобы пользователь мог задать цвет какого-то объекта, например, цвет фона компонента **Memor1**, то это можно реализовать оператором

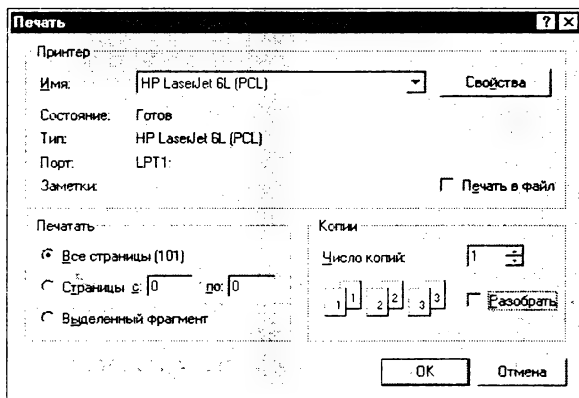
```
If ColorDialog1.Execute then
    Memor1.Color:=ColorDialog1.Color;
```

8.6 Диалоги печати и установки принтера — компоненты **PrintDialog** и **PrinterSetupDialog**

Компонент **PrintDialog** вызывает диалоговое окно печати, представленное на рис. 8.10. В нем пользователь может выбрать принтер и установить его свойства, указать число копий и после-

Рис. 8.10

Диалоговое окно настройки печати



довательность их печати, печатать в файл или непосредственно на принтер, выбрать печатаемые страницы или печатать только выделенный фрагмент.

Компонент **PrintDialog** не осуществляет печать. Он только позволяет пользователю задать атрибуты печати. А сама печать должна осуществляться программно с помощью объекта **Printer** или иным путем (о способах печати см. раздел 4.7).

Рассмотренные ранее диалоговые компоненты возвращали одно свойство — имя файла, цвет, или один объект — **Font**, содержащий множество свойств. В отличие от них компонент **PrintDialog** возвращает ряд свойств, характеризующих выбранные пользователем установки. Это следующие свойства:

PrintRange	Показывает выбранную пользователем радиокнопку из группы Печатать: prAllPages — выбрана кнопка Все страницы, prSelection — выбрана кнопка Страницы с ... по ..., prPageNums — выбрана кнопка Страницы
FromPage	Показывает установленную пользователем начальную страницу в окне Страницы с ... по ...
ToPage	Показывает установленную пользователем конечную страницу в окне Страницы с ... по ...
PrintToFile	Показывает, выбран ли пользователем индикатор Печать в файл
Copies	Показывает установленное пользователем число копий
Collate	Показывает, выбран ли пользователем индикатор Разобрать

Перед вызовом диалога желательно определить, сколько страниц в печатаемом тексте, и задать параметры **MaxPage** и **MinPage** — максимальный и минимальный номера страниц. В противном случае пользователю в диалоговом окне не будет доступна кнопка Страницы с ... по Кроме того следует определить множество опций в свойстве **Options**:

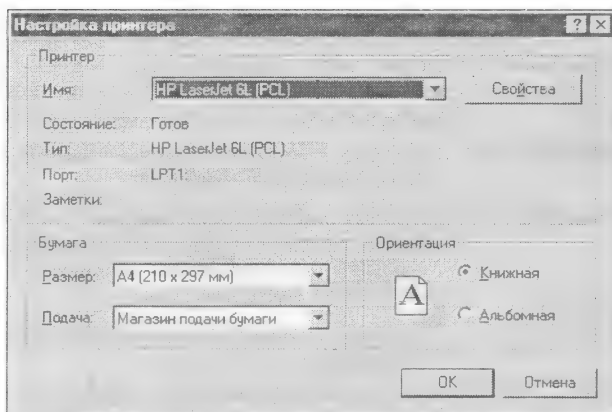
poDisablePrintToFile	Запретить доступ к индикатору Печать в файл. Эта опция работает только при включенной опции poPrintToFile
poHelp	Отображать в диалоговом окне кнопку Справка. Опция может не работать для некоторых версий Windows 95/98

poPageNums	Сделать доступной радиокнопку Страницы, позволяющую пользователю задавать диапазон печатаемых страниц
poPrintToFile	Отображать в диалоговом окне кнопку Печать в файл
poSelection	Сделать доступной кнопку Выделение, позволяющую пользователю печатать только выделенный текст
poWarning	Выдавать замечания, если пользователь пытается послать задачу на неустановленный принтер

Теперь остановимся на компоненте **PrinterSetupDialog**, вызывающем диалоговое окно установки принтера, представленное на рис. 8.11. Это единственный диалоговый компонент, не имеющий никаких специфических свойств, которые надо было бы устанавливать или читать. Диалог выполняет операции по установке принтера, на котором будет производиться печать, и задании его свойств. Этот диалог не возвращает никаких параметров.

Рис. 8.11

Диалоговое окно
установки принтера



8.7 Диалоги поиска и замены текста — компоненты FindDialog и ReplaceDialog

Компоненты **FindDialog** и **ReplaceDialog**, вызывающие диалоги поиска и замены фрагментов текста (рис. 8.12 и 8.13), очень похожи и имеют одинаковые свойства, кроме одного, задающего заменяющий текст в компоненте **ReplaceDialog**. Такое сходство

не удивительно, поскольку **ReplaceDialog** — производный класс от **FindDialog**.

Рис. 8.12

Диалоговое окно поиска фрагмента текста

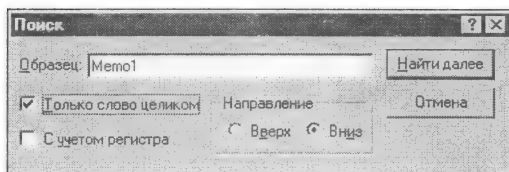
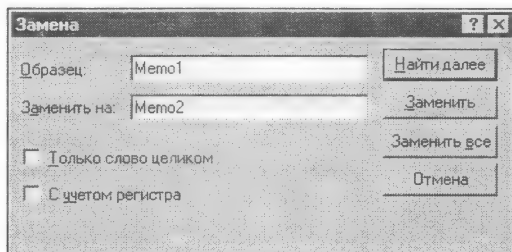


Рис. 8.13

Диалоговое окно замены фрагмента текста



Компоненты имеют следующие основные свойства:

FindText	Текст, заданный пользователем для поиска или замены. Программно может быть установлен как начальное значение, предлагаемое пользователю
ReplaceText	Только в компоненте ReplaceDialog — текст, который должен заменять FindText
Position	Позиция левого верхнего угла диалогового окна, заданная типом TPoint — записью, содержащей поля X (экранная координата по горизонтали) и Y (экранная координата по вертикали)
Left	Координата левого края диалогового окна, то же, что Position.X
Top	Координата верхнего края диалогового окна, то же, что Position.Y
Options	Множество опций

Последний параметр **Options** — может содержать следующие свойства:

frDisableMatchCase	Делает недоступным индикатор С учетом регистра в диалоговом окне
frDisableUpDown	Делает недоступными в диалоговом окне кнопки Вверх и Вниз группы Направление, определяющие направление поиска
frDisableWholeWord	Делает недоступным индикатор Только слово целиком в диалоговом окне
frDown	Выбирает кнопку Вниз группы Направление при открытии диалогового окна. Если эта опция не установлена, то выбирается кнопка Вверх
frFindNext	Эта опция включается автоматически, когда пользователь в диалоговом окне щелкает на кнопке Найти далее, и выключается при закрытии диалога
frHideMatchCase	Удаляет индикатор С учетом регистра из диалогового окна
frHideWholeWord	Удаляет индикатор Только слово целиком из диалогового окна
frHideUpDown	Удаляет кнопки Вверх и Вниз из диалогового окна
frMatchCase	Этот флаг включается и выключается, если пользователь включает и выключает опцию С учетом регистра в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена
frReplace	Применяется только для ReplaceDialog . Этот флаг устанавливается системой, чтобы показать, что текущее (и только текущее) найденное значение FindText должно быть заменено значением ReplaceText
frReplaceAll	Применяется только для ReplaceDialog . Этот флаг устанавливается системой, чтобы показать, что все найденные значения FindText должны быть заменены значениями ReplaceText
frShowHelp	Задаёт отображение кнопки Справка в диалоговом окне
frWholeWord	Этот флаг включается и выключается, если пользователь включает и выключает опцию Только слово целиком в диалоговом окне. Можно установить эту опцию по умолчанию во время проектирования, чтобы при открытии диалога она была включена

Сами по себе компоненты **FindDialog** и **ReplaceDialog** не осуществляют ни поиска, ни замены. Они только обеспечивают интерфейс с пользователем. А поиск и замену надо осуществлять программно. Для этого можно пользоваться событием **OnFind**, происходящим, когда пользователь нажал в диалоге кнопку Найти далее, и событием **OnReplace**, возникающим, если пользователь нажал кнопку Заменить или Заменить все. В событии **OnReplace** узнать, какую именно кнопку нажал пользователь, можно по значениям флагов **frReplace** и **frReplaceAll**.

Поиск заданного фрагмента легко проводить, пользуясь функцией Object Pascal **Pos**, которая определена в модуле **System** следующим образом:

```
function Pos(Substr: string; S: string): Byte;
```

где **S** — строка, в которой ищется фрагмент текста, а **Substr** — искомый фрагмент. Функция возвращает позицию первого символа первого вхождения искомого фрагмента в строку. Если **Substr** в **S** не найден, возвращается 0.

Для организации поиска нам потребуется еще две функции: **Copy** и **AnsiLowerCase**. Первая из них определена как:

```
function Copy(S: string; Index, Count: Integer): string;
```

Она возвращает фрагмент строки **S**, начинающийся с позиции **Index** и содержащий число символов, не превышающее **Count**.

Функция **AnsiLowerCase**, определенная как

```
function AnsiLowerCase(const S: string): string;
```

возвращает строку символов **S**, переведенную в нижний регистр.

Теперь мы можем рассмотреть пример организации поиска. Пусть в вашем приложении имеется компонент **Memo1** и при выборе раздела меню **MFind** вы хотите организовать поиск в тексте, содержащемся в **Memo1**. Для упрощения задачи исключим опцию поиска только целых слов и опцию поиска вверх от положения курсора.

Программа, реализующая поиск, может иметь следующий вид:

```
var SPos:integer;
...
procedure TForm1.MFindClick(Sender: TObject);
begin
  {запоминание позиции курсора}
  SPos:=Memo1.SelStart;
  with FindDialog1 do begin
    {начальное значение текста поиска — текст,
     выделенный в Memo1}
    FindText:=Memo1.SelText;
    {позиционирование окна диалога внизу Memo1}
```

```
Position:=Point(Form1.Left,Form1.Top+
                Memol.Top+Memol.Height);
{удаление из диалога кнопок «Вверх», «Вниз»,
 «Только слово целиком»}
Options:=Options + [frHideUpDown,frHideWholeWord];
{выполнение}
Execute;
end;
end;

procedure TForm1.FindDialog1Find(Sender: TObject);
begin
  with FindDialog1 do begin
    if frMatchCase in Options
      {поиск с учетом регистра}
      then Memol.SelStart:=Pos(FindText,
                              Copy(Memol.Lines.Text,SPos+1,
                                   Length(Memol.Lines.Text)))+SPos-1
      {поиск без учета регистра}
    else Memol.SelStart:=Pos(AnsiLowerCase(FindText),
                              AnsiLowerCase(Copy(Memol.Lines.Text,SPos+1,
                                   Length(Memol.Lines.Text))))+SPos-1;
    if Memol.SelStart>=SPos
      then begin
        {выделение найденного текста}
        Memol.SelLength:=Length(FindText);
        {изменение начальной позиции поиска}
        SPos:=Memol.SelStart+Memol.SelLength+1;
      end
    else
      if MessageDlg(
        'Текст "'+FindText+'" не найден.
        Продолжать диалог?',
        mtConfirmation,mbYesNoCancel,0) <> mrYes
      then CloseDialog;
    end;
    Memol.SetFocus;
  end;
end;
```

В программе вводится переменная **SPos**, сохраняющая позицию, начиная с которой надо проводить поиск.

Процедура **MFindClick** вызывает диалог, процедура **FindDialog1Find** обеспечивает поиск с учетом или без учета регистра в зависимости от флага **frMatchCase**. После нахождения очередного вхождения искомого текста этот текст выделяется в окне **Memol** и управление передается этому окну редактирования. Затем при нажатии пользователем в диалоговом окне кнопки Найти далее, поиск продолжается в оставшейся части текста. Если искомый текст не найден, делается запрос пользователю о

продолжении диалога. Если пользователь не ответил на этот запрос положительно, то диалог закрывается методом **CloseDialog**.

В дополнение к приведенному тексту полезно в обработчики событий **OnClick** и **OnKeyUp** компонента **Memo1** ввести операторы

```
SPos:=Memo1.SelStart;
```

Это позволяет пользователю во время диалога изменить положение курсора в окне **Memo1**. Это новое положение сохранится в переменной **SPos** и будет использовано при продолжении поиска.

При реализации команды **Заменить** приведенные выше процедуры можно оставить теми же самыми, заменив в них **FindDialog1** на **ReplaceDialog1**. Дополнительно можно написать процедуру обработки события **OnReplace** компонента **ReplaceDialog1**:

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
begin
  if Memo1.SelText<>''
  then Memo1.SelText:=ReplaceDialog1.ReplaceText;
  if frReplaceAll in ReplaceDialog1.Options
  then ReplaceDialog1Find(Self);
end;
```

Этот код производит замену выделенного текста и, если пользователь нажал кнопку **Заменить все**, то продолжается поиск вызовом уже имеющейся процедуры поиска **ReplaceDialog1Find**. Если же пользователь нажал кнопку **Заменить**, то производится только одна замена и для продолжения поиска пользователь должен нажать кнопку **Найти далее**.

9.1 Общая характеристика

Имеется несколько компонентов, которые сами по себе не выполняют никаких действий, но помогают организовать взаимодействие компонентов в приложении и обмен информацией между приложениями. Это компоненты, представленные в таблице 9.1. Компоненты **ActionList** и **ImageList** организуют управление внутри приложения и позволяют сделать его код более четким и понятным, отвечающим принципам объектно-ориентированного программирования.

Компоненты **OLEContainer**, **DDEClientConv**, **DDEClientItem** и **TDDEServerItem** организуют взаимодействие одновременно выполняемых приложений по двум различным технологиям: OLE и DDE. Вопросы взаимодействия приложений требуют развернутых пояснений, выходящих за рамки данной книги. Поэтому компоненты, связанные с технологиями OLE и DDE, рассматриваться далее не будут.

К компонентам, организующим управление приложением, можно также отнести объекты **Application** — приложение и **Screen** — экран. Эти объекты отсутствуют в палитре компонентов, но всегда создаются автоматически в каждом приложении. Некоторые характеристики этих объектов также будут рассмотрены в данной главе.

Таблица 9.1. Компоненты организации управления приложением

Пиктограмма	Компонент	Страница	Описание
	ActionList (список событий)	Standard	Обеспечивает диспетчеризацию событий компонентов
	ImageList (список изображений)	Win32	Предназначен для работы со списками изображений одинакового размера в меню, инструментальных панелях и т.п.

Пикто- грамма	Компонент	Страница	Описание
	ApplicationEvents (события приложения)	Additional	Перехватывает события приложения
	OLEContainer (контейнер OLE)	System	Используется при создании области клиента для объекта OLE
	DDEClientConv (диалог с сервером DDE)	System	Используется клиентом DDE для организации диалога с сервером DDE
	DDEClientItem (данные, передаваемые серверу DDE)	System	Используется для определения данных клиента, передаваемых в диалоге серверу DDE
	TDDEServerItem (данные, передаваемые клиенту DDE)	System	Используется для определения данных сервера, передаваемых клиенту DDE в течение диалога

9.2 Диспетчеризация событий — компоненты, связанные с **ActionList**

Начиная с Delphi 4 появился инструментарий, который, не добавляя никаких принципиально новых возможностей, позволяет систематизировать и упорядочить разработку объектно-ориентированных приложений. К тому же грамотное его использование позволяет сэкономить немало времени при проектировании. Для организации диспетчеризации событий введены следующие понятия.

Действие (action) — реализация некоторого поведения, являющегося реакцией на поступок пользователя, такой, как щелчок на кнопке или на разделе меню — *инициаторе действия*. Разработан класс **TAction** и ряд его наследников, реализующих многие стандартные действия, характерные для приложений Windows.

Список действий — компонент типа **ActionList**, содержащий предусмотренные в приложении действия. Это интерфейс разработчика, упорядочивающий его работу с действиями в процессе проектирования.

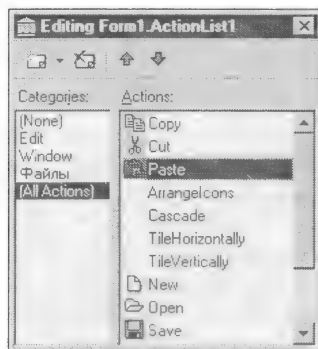
Редактор связей — объект типа **TActionLink**, который поддерживает связь между действиями и инициаторами действий. Редактор связей определяет, какое действие должно выполняться для данного инициатора.

Цель действия — объект, в котором отражается результат действия. Это может быть окно отображения или редактирования текста, набор данных и т.п.

В начале проектирования приложения разработчик должен представить себе список тех действий, которые может осуществлять пользователь. Конечно, в процессе проектирования этот список будет пополняться и изменяться, но некоторое начальное приближение очень полезно продумать заранее. Практическая реализация составленного вами списка действий может начинаться с переноса на проектируемую форму невидимого компонента **ActionList**, расположенного на странице библиотеки Standard. Сделав на этом компоненте двойной щелчок, вы попадаете в редактор действий (рис. 9.1), позволяющий вводить и упорядочивать действия.

Рис. 9.1

Окно редактора действий

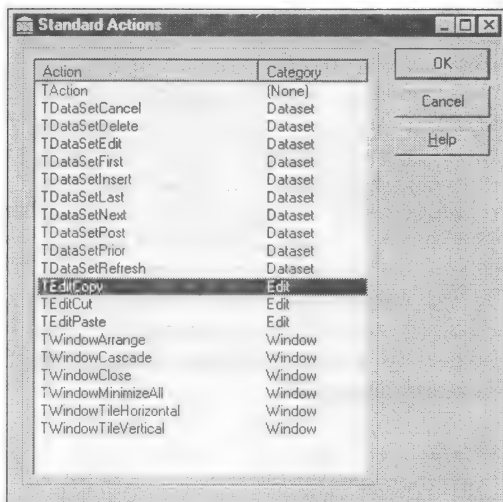


Щелчок правой кнопкой мыши или щелчок на маленькой кнопке со стрелкой вниз правее первой быстрой кнопки окна редактирования позволит вам выбрать одну из команд: New Action (новое действие) или New Standard Action... (новое стандартное действие). Первая из них относится к вводу нового действия любого типа. По умолчанию эти действия будут получать имена **Action1**, **Action2** и т.д. Вторая команда открывает окно (рис. 9.2), в котором вы можете выбрать необходимое вам стандартное действие (или сразу несколько действий). После этого в правом окне (Actions) редактора появятся имена выбранных действий, а в левом (Categories) — категории действий.

Каждое действие, которое вы внесли в список — это объект типа **TAction** для нестандартных действий или других производных типов для стандартных. Выбрав в окнах редактора ту или иную категорию или [AllActions] (все категории), а в правом — конкретное действие, вы можете увидеть в Инспекторе Объектов

Рис. 9.2

Окно выбора стандартных действий



его свойства и события. Вы можете установить свойство **Name** (имя) — оно появится в правом окне редактора свойств и будет в дальнейшем фигурировать в заголовках обработчиков событий. Можете также установить надпись (**Caption**), которая далее будет появляться в инициаторах действия — кнопках, разделах меню и т.д. Можете задать быстрые клавиши (**ShortCut**), надписи на ярлычках подсказок и в строке состояний (**Hint**), идентификатор темы контекстной справки (**HelpContext**) и другие обычные для многих компонентов свойства.

Можно для каждого или некоторых действий указать свойство **ImageIndex**, которое является индексом (начиная с 0) изображения, соответствующего данному действию в отдельном компоненте списка изображений **ImageList** (см. раздел 9.3). Этот индекс передается в дальнейшем компонентам, связанным с данным событием — разделам меню, кнопкам. Если в свойстве **Images** компонента **ActionList** указать имя списка, размещенного на форме и заполненного изображениями, то эти изображения появятся также в окне редактора действий (рис. 9.1).

Свойство **Category** (категория) не имеет отношения к выполнению приложения. Задание категории просто позволяет в процессе проектирования сгруппировать действия по их назначению. Вы можете для каждого действия выбрать категорию из выпадающего списка, или написать имя новой категории и отнести к ней какие-то действия. Например, на рис. 9.1 видна введенная пользователем категория **Файлы**, к которой отнесены действия, связанные с меню **Файл**.

На странице событий Инспектора Объектов для каждого действия определено три события: **OnExecute**, **OnUpdate** и **OnHint**.

Событие **OnExecute** возникает в момент, когда пользователь инициализировал действие, например, щелкнув на компоненте (разделе меню, кнопке), связанном с данным действием. Обработчик этого события должен содержать процедуру, реализующую данное действие. Например, обработчик события **OnExecute** действия **Exit** может в простейшем случае иметь вид

```
procedure TForm1.ExitExecute(Sender: TObject);  
begin  
  Close;  
end;
```

а в более сложных случаях может содержать проверку возможности закрыть приложение, запросы пользователю и т.д. Одним из преимуществ использования действий является то, что заголовки обработчиков приобретают осмысленный характер и код становится более прозрачным. Действительно, гораздо понятнее заголовков **ExitExecute**, чем, например, **Button7Click** или **N14Click** (попробуйте найти в вашем большом приложении, где эта кнопка **Button7** или раздел меню **N14**). В результате вы избавляетесь от необходимости давать осмысленные имена кнопкам и разделам меню, т.е. облегчаете свою работу с компонентами.

Событие **OnUpdate** периодически возникает в промежутках между действиями. Возникновение этих событий прекращается только во время реализации события или во время, когда пользователь ничего не делает и компьютер находится в состоянии ожидания действий. Обработчик события **OnUpdate** может содержать какие-то настройки, подготовку ожидаемых дальнейших действий или выполнение каких-то фоновых операций.

Событие **OnHint** возникает в момент, когда на экране отображается ярлычок подсказки в результате того, что пользователь задержал курсор мыши над компонентом, инициализирующим событие.

Наличие в объекте действия событий **OnUpdate** и **OnHint** обогащает ваши возможности по проектированию приложения. Без этого объекта подобные события отсутствуют и их при необходимости приходится моделировать более сложными приемами.

Связь объектов действий с конкретными инициализаторами действий — управляющими элементами типа кнопок, разделов меню и т.д., осуществляется через свойство **Action**, имеющееся у всех управляющих элементов. Поместите на вашу форму кнопку, и вы увидите в Инспекторе Объектов это свойство. Раскройте его выпадающий список и выберите из него действие, которое вами было предварительно описано. Вы сможете заметить, что

после этого в кнопку перенесутся такие свойства объекта действия, как **Caption**, **Hint** и др., и что в ее событие **OnClick** подставится обработчик, предусмотренный вами для данного действия. Если далее вы введете в приложение меню с разделом, соответствующим тому же действию, и укажете в нем то же значение свойства **Action**, то свойства и обработчики событий объекта действия будут перенесены и на этот раздел меню. Вам не придется повторно задавать значение **Hint**, **Caption** и др.

Подобная связь между свойствами объекта действия и свойствами управляющих элементов выполняется классом **TActionLink** и его наследниками. Передаются в компоненты такие свойства действия, как **Caption**, **Checked**, **Enabled**, **HelpContext**, **Hint**, **ImageIndex**, **ShortCut**, **Visible**. Впрочем, в любом компоненте разработчик может изменить переданное в него свойство. Обратной связи **TActionLink** с компонентами нет, так что эти изменения будут локальными и не отразятся на других компонентах. Если же требуется изменить свойства всех связанных с одним действием компонентов, надо изменять свойство объекта действия. Это облегчает программное управление компонентами, связанными с одним и тем же действием. Например, если в какой-то момент вам надо сделать недоступными (или, наоборот, доступными) два компонента — кнопку **Button3** и раздел меню **N5**, связанные с одним событием (назовем его объект **Do**), то при отсутствии централизованной диспетчеризации событий через **ActionList** вам пришлось бы писать два оператора:

```
Button3.Enabled := false;  
N5.Enabled := false;
```

а при наличии объекта **Do** — всего один:

```
Do.Enabled := false;
```

Дело не только в экономии кода, но и в его прозрачности, понятности его как для вас самих, так и для тех, кому придется, может быть, в дальнейшем сопровождать ваше приложение.

9.3 Список изображений — компонент **ImageList**

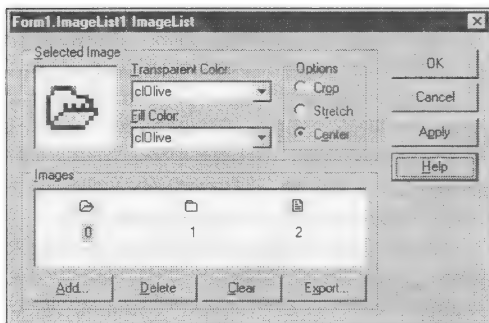
Компонент **ImageList** представляет собой набор изображений одинаковых размеров, на которые можно ссылаться по индексам, начинающимся с 0. Во многих рассмотренных ранее компонентах (меню, списках и др.) встречались свойства, представляющие собой ссылки на компонент **ImageList**. Этот компонент позволяет организовать эффективное и экономное управления множеством пиктограмм и битовых матриц. Он может включать в

себя монохромные битовые матрицы, содержащие маски для отображения прозрачности рисуемых изображений.

Изображения в компонент **TImageList** могут быть загружены в процессе проектирования с помощью редактора списков изображений. Окно редактора, представленное на рис. 9.3, вызывается двойным щелчком на компоненте **TImageList** или щелчком правой кнопки мыши и выбором команды контекстного меню **ImageList Editor**.

Рис. 9.3

Окно редактора списков изображений



В окне редактора списков изображений вы можете добавить в списки изображения, пользуясь кнопкой **Add**, удалить изображение из списка кнопкой **Delete**, очистить весь список кнопкой **Clear**. При добавлении изображения в список открывается обычное окно открытия файлов изображений, в котором вы можете выбрать интересующий вас файл. Только учтите, что размер всех изображений в списке должен быть одинаковым. Как правило, это размер, используемый для пиктограмм в меню, списках, кнопках. При добавлении в список изображений для кнопок надо иметь в виду, что они часто содержат не одно, а два и более изображений (см. раздел 5.2). В этих случаях при попытке добавить изображение задается вопрос: «Bitmap dimensions for ... are greater then imagelist dimensions. Separate into ... separate bitmaps?» (Размерность изображения ... больше размерности списка. Разделить на ... отдельных битовых матрицы?). Если вы ответите отрицательно, то все изображения уменьшатся в горизонтальном размере и лягут как одно изображение. Использовать его в дальнейшем будет невозможно. Поэтому на заданный вопрос надо отвечать положительно. Тогда загружаемая битовая матрица автоматически разделится на отдельные изображения и потом вы можете удалить те из них, которые вам не нужны, кнопкой **Delete**.

Как видно из рис. 9.3, каждое загруженное в список изображение получает индекс. Именно на эти индексы впоследствии вы

можете ссылаться в соответствующих свойствах разделов меню, списков, кнопок и т.д., когда вам надо загрузить в них то или иное изображение. Изменить последовательность изображений в списке вы можете просто перетаскив изображение мышью на новое место.

В редакторе списков изображений вы можете, выделив то или иное изображение, установить его свойства: `Transparent Color` и `Fill Color`. Но это можно делать только в том сеансе работы с редактором списков изображений, в котором загружено данное изображение. Для изображений, загруженных в предыдущих сеансах, изменение этих свойств невозможно.

Свойство `Transparent Color` определяет цвет, который используется в маске для прозрачного рисования изображения. По умолчанию это цвет левого нижнего пикселя изображения. Для пиктограмм данное свойство устанавливается в `clNone`, поскольку пиктограммы уже маскированы.

Свойство `Fill Color` определяет цвет, используемый для заполнения пустого пространства при перемещении и центрировании изображения. Для пиктограмм данное свойство устанавливается в `clNone`.

Группа радиокнопок `Options` определяет способ размещения изображения битовой матрицы с размерами, не соответствующими размерам, принятым в списке:

Crop	Отображается часть изображения, помещающаяся в размер списка, начиная с левого верхнего угла
Stretch	Размеры изображения изменяются, становясь равными размерам списка. При этом возможны искажения
Center	Изображение центрируется, а если его размер больше размера списка, то не помещающиеся области отсекаются

Теперь рассмотрим основные свойства `TImageList`:

Свойство	Тип	Описание
Height	Integer	Высота изображений в списке
Width	Integer	Ширина изображений в списке
AllocBy	Integer	Определяет количество изображений, на которое увеличивается список для добавления новых изображений
Count	Integer	Определяет число изображений в списке. Свойство только для чтения

Остальные свойства определяют цвета и способы рисования изображения.

9.4 Приложение — компонент **ApplicationEvents** и объект **Application**

В каждом приложении автоматически создается объект **Application** типа **TApplication** — приложение. Этот компонент отсутствует в палитре библиотеки, вероятно, только потому, что он всегда один в приложении. **Application** имеет ряд свойств, методов, событий, характеризующих приложение в целом.

Рассмотрим сначала некоторые свойства **Application**. Булево свойство **Active** (только для чтения) характеризует активность приложения. Оно равно **true**, если форма приложения находится в фокусе. Если же пользователь переключился на работу с другим приложением, свойство **Active** равно **false**.

Свойство **ExeName** является строкой, содержащей имя выполняемого файла с полным путем к нему. Это свойство удобно использовать, чтобы определить каталог, из которого запущено приложение и который может содержать другие файлы (настройки, документы, базы данных и т.п.), связанные с приложением. Выражение **ExtractFilePath(Application.ExeName)** дает этот каталог. Обычно свойство **ExeName** тождественно функции **ParamStr(0)**, возвращающей нулевой параметр командной строки — имя файла с путем.

Свойство **Title** определяет строку, которая появляется около пиктограммы свернутого приложения. Если это свойство не изменяется во время выполнения, то оно равно опции **Title**, задаваемой во время проектирования на странице **Application** окна опций проекта (команда **Project | Options**). Свойство может изменяться программно, например, изменяя надпись в зависимости от режима работы приложения.

Свойство **MainForm** типа **TForm** определяет главную форму приложения. Булево свойство **ShowMainForm** определяет, должна ли главная форма быть видимой в момент запуска приложения на выполнение. По умолчанию оно равно **true**, что обеспечивает видимость главной формы в момент начала работы приложения. Если же установить в головном файле проекта **Application.ShowMainForm** равным **false** до вызова метода **Application.Run** и если при этом свойство **Visible** главной формы тоже равно **false**, то главная форма в первый момент будет невидимой.

Свойство **HelpFile** указывает файл справки, который используется в приложении в данный момент как файл по умолчанию.

Если это свойство не изменяется во время выполнения, то оно равно опции **Help File**, задаваемой во время проектирования на странице **Application** окна опций проекта (команда **Project | Options**). Свойство можно изменять программно, назначая в зависимости от режима работы приложения тот или иной файл справки.

Ряд свойств объекта **Application** определяет ярлычки подсказок компонентов приложения. Свойство **Hint** содержит текст подсказки **Hint** того визуального компонента или раздела меню, над которым в данный момент перемещается курсор мыши. Смена этого свойства происходит в момент события **OnHint**, которое будет рассмотрено позднее. Во время этого события текст подсказки переносится из свойства **Hint** компонента, на который переместился курсор мыши, в свойство **Hint** объекта **Application**. Свойство **Application.Hint** можно использовать для отображения этой подсказки или для установки и отображения в полосе состояния текста, характеризующего текущий режим приложения.

Свойство **HintColor** типа **TColor** определяет цвет фона окна ярлычка. По умолчанию это цвет **clInfoBk**, но его значение можно изменять программно. Свойство **HintPause** определяет задержку появления ярлычка в миллисекундах после переноса курсора мыши на очередной компонент (по умолчанию 500 миллисекунд или половина секунды). Свойство **HintHidePause** аналогичным образом определяет интервал времени, после которого ярлычок становится невидимым (по умолчанию 2500 миллисекунд или две с половиной секунды). Свойство **HintShortPause** определяет аналогичным образом задержку перед появлением нового ярлычка, если в данный момент отображается другой ярлычок (по умолчанию 50 миллисекунд). Это свойство позволяет предотвратить неприятное мерцание, если пользователь быстро перемещает курсор мыши над разными компонентами.

Теперь остановимся на некоторых методах объекта **Application**. Методы **Initialize** — инициализация проекта, и **Run** — запуск выполнения приложения, включаются в каждый проект автоматически — вы можете это увидеть в головном файле проекта, если выполните команду **Project | View Source**. Там же вы можете увидеть применение метода создания форм **CreateForm** для всех автоматически создаваемых форм проекта. Если же в вашем проекте есть форма, которая исключена из списка автоматически создаваемых (команда **Project | Options** и соответствующая установка на странице **Forms**), то когда эта форма вам потребуется, вы должны будете вызвать этот метод:

```
procedure CreateForm(FormClass: TFormClass;  
    var Reference);
```

где **FormClass** — класс создаваемой формы, **Reference** — ссылка на создаваемый объект (его имя). Например:

```
Application.CreateForm(TForm2, Form2);
```

Метод **Terminate** завершает выполнение приложения. Если вам надо завершить приложение из главной формы, то вместо метода **Application.Terminate** вы можете использовать метод **Close** главной формы. Но если вам надо закрыть приложение из какой-то вторичной формы, например, из диалога, то надо применять метод **Application.Terminate**.

Метод **Minimize** сворачивает приложение, помещая его пиктограмму в полосу задач Windows.

Ряд методов связан с работой со справочными файлами. Выше уже говорилось о свойстве **HelpFile**, указывающем текущий файл справки. Метод **HelpContext**:

```
function HelpContext(Context: THelpContext): Boolean;
```

вызывает переход в файл справки на тему с идентификатором **Context**. Это идентификатор, который при проектировании справки поставлен в соответствие некоторой теме. Метод **HelpJump**:

```
function HelpJump(const JumpID: string): Boolean;
```

выполняет аналогичные действия, но его параметр **JumpID** — не идентификатор темы, а имя соответствующей темы в файле справки, задаваемое в нем сноской #.

Метод **HelpCommand**:

```
function HelpCommand(Command: Word;  
Data: Longint): Boolean;
```

позволяет выполнить указанную параметром **Command** команду API WinHelp с параметром **Data**. Метод генерирует событие **OnHelp** активной формы или приложения, а затем выполняет указанную команду WinHelp. Полный список команд WinHelp вы можете найти в теме WinHelp справочного файла **win32.hlp**, расположенного в каталоге ...\\Program Files\\Common Files\\Borland Shared\\MSHelp. Приведем только некоторые из них. Команда **HELP_CONTENTS** с параметром 0 отображает окно Содержание справки. Команда **HELP_INDEX** с параметром 0 отображает окно Указатель справки. Команда **HELP_CONTEXT** с параметром, равным идентификатору темы, отображает тему с заданным идентификатором (это тождественно рассмотренному ранее методу **HelpContext**). Команда **HELP_CONTEXTPOPUP** с параметром, равным идентификатору темы, делает то же самое, но отображает тему во всплывающем окне.

В классе **TApplication** имеется еще немало методов, но часть из них используется в явном виде очень редко (вы можете по-

смотреть их во встроенной справке Delphi), а часть будет рассмотрена ниже при обсуждении событий объекта **Application**. Хотелось бы только обратить внимание читателя на очень полезный метод **MessageBox**, позволяющий вызывать диалоговое окно с указанным текстом, указанным заголовком и русскими надписями на кнопках (в русифицированных версиях Windows). Это наиболее удачный полностью русифицируемый стандартный диалог, но его рассмотрение, к сожалению, выходит за рамки данной книги. См. о нем в следующих книгах серии «Все о Delphi».

В классе **TApplication** определено множество событий, которые очень полезны для организации приложения. Ранее для использования этих событий было необходимо вводить соответствующие обработчики и указывать на них объекту **Application** специальными операторами. В Delphi 5 введен компонент **ApplicationEvents**, существенно облегчивший эту задачу. Этот компонент перехватывает события объекта **Application** и, следовательно, обработчики этих событий теперь можно писать как обработчики событий невизуального компонента **ApplicationEvents**. На каждой форме приложения можно разместить свой компонент **ApplicationEvents**. События объекта **Application** будут передаваться всем этим компонентам. Если вы хотите, чтобы событие передавалось прежде всего какому-то одному из них, примените к нему метод **Activate**, который поставит его в начало очереди компонентов **ApplicationEvents**. Если же вы при этом не хотите, чтобы другие компоненты **ApplicationEvents** получали события, примените к привилегированному компоненту метод **CancelDispatch**. Тогда после обработки события в данном компоненте другие компоненты **ApplicationEvents** вообще не будут реагировать на эти события.

Во многие обработчики событий компонента **ApplicationEvents** передается по ссылке параметр **Handled**. По умолчанию его значение равно **false**. Если вы обработали соответствующее событие и не хотите, чтобы оно далее обрабатывалось другими компонентами **ApplicationEvents**, надо в обработчике установить **Handled = true**. Если же вы оставите **Handled = false**, то событие будут пытаться обрабатывать другие компоненты **ApplicationEvents** (если они есть). Если событие так и останется необработанным, то его будет пытаться обработать активный компонент, а если не обработает — то активная форма. Предотвратить обработку события другими компонентами можно, используя описанный ранее метод **CancelDispatch**.

Ниже приведена таблица событий компонента **ApplicationEvents** с их краткими описаниями.

Событие	Описание
OnActionExecute	Возникает при выполнении (Execute) некоторого действия, объявленного в компоненте ActionList , но не обработанного им (не написан соответствующий обработчик). Инициализация этого события может быть, например, выполнена методом Application.ExecuteAction(<имя действия>) . Если событие не обработано в ActionList , то оно может быть обработано на уровне приложения. В обработчик OnActionExecute передается параметр Action — действие и по ссылке передается параметр Handled (см. выше).
OnActionUpdate	Возникает при обновлении (Update) некоторого действия, объявленного в компоненте ActionList , но не обработанного им (не написан соответствующий обработчик). Если событие не обработано в ActionList , то оно может быть обработано на уровне приложения. В обработчик OnActionUpdate передается параметр Action — действие и по ссылке передается параметр Handled (см. выше).
OnActivate	Возникает, когда приложение становится активным. Это происходит при начале выполнения и в случаях, когда пользователь, перейдя к другим приложениям, вернулся в данное. Если это событие обработано в ApplicationEvents , то предотвратить дальнейшую его обработку можно методом CancelDispatch .
OnDeactivate	Возникает перед тем моментом, когда приложение перестает быть активным (пользователь переключается на другое приложение). Если событие обработано в ApplicationEvents , то предотвратить дальнейшую его обработку можно методом CancelDispatch .
OnException	Возникает, когда в приложении сгенерировано исключение, которое нигде не перехвачено. В обработчик передается параметр Sender — источник исключения, и параметр E типа Exception — объект исключения. Параметр E помогает определить тип исключения. Например, if (E is EDivByZero) then В обработчике события OnException вы можете предусмотреть нестандартную обработку исключений на уровне приложения, например, русифицировать стандартные сообщения об исключениях и дать пользователю какие-то рекомендации. Учтите, что введение вами обработчика OnException отключит стандартную реакцию приложения на исключительные ситуации.

Событие	Описание
OnHelp	Возникает при запросе приложением справки. Это событие возникает, в частности, при выполнении рассмотренных ранее методов приложения HelpContext , HelpJump и HelpCommand . Обработчик может использоваться для каких-то подготовительных операций, например, для задания файла справки (параметр Application.HelpFile). В обработчик передаются параметры — Command команда API WinHelp (см. выше описание HelpCommand), Data — параметр этой команды и по ссылке передается булев параметр CallHelp . Если его установить в true , то после завершения обработчика будет вызван WinHelp, в противном случае вызова WinHelp не будет.
OnHint	Возникает в момент, когда курсор мыши начинает перемещаться над компонентом или разделом меню, в котором определено свойство Hint . При этом свойство Hint компонента переносится в свойство Hint приложения (Application.Hint) и в обработчике данного события может отображаться, например, в строке состояния.
OnIdle	Возникает, когда приложение начинает простаивать, ожидая, например, действий пользователя. В обработчик передается по ссылке булев параметр Done , который по умолчанию равен true . Если оставить его без изменения, то по окончании работы обработчика данного события будет вызвана функция WaitMessage API Windows, которая займется в период ожидания другими приложениями. Если задать Done = false , то эта функция вызываться не будет. Это может снизить производительность Windows.
OnMessage	Возникает, когда приложение получает сообщение Windows (но не переданное функцией SendMessage API Windows). В обработчике события OnMessage можно предусмотреть нестандартную (отличную от определенной в TApplication) обработку сообщения. В обработчик передается параметр Msg — полученное сообщение и по ссылке передается параметр Handled (см. выше). Учтите, что в Windows передаются тысячи сообщений в секунду, так что обработчик OnMessage может серьезно снизить производительность приложения.
OnMinimize	Возникает при сворачивании приложения.
OnRestore	Возникает при восстановлении ранее свернутого приложения.

Событие	Описание
OnShortCut	Возникает при нажатии пользователем клавиши. Событие возникает до того, как возникло стандартное событие OnKeyDown компонента или формы. Обработчик позволяет предусмотреть нестандартную реакцию на нажатие какой-то клавиши. В него передается параметр сообщения Windows Msg , поле CharCode которого (Msg.CharCode) содержит виртуальный код нажатой клавиши. Передается также по ссылке параметр Handled . Если задать ему значение true , то стандартные события OnKeyDown , OnKeyPress , OnKeyUp не наступят.
OnShowHint	Возникает, когда приложение собирается отобразить ярлычок с текстом подсказки Hint . В обработчик передается по ссылке параметр HintStr — первая часть свойства Hint компонента, ярлычок которого должен отображаться. В обработчике этот текст можно изменить. Так же по ссылке передается параметр CanShow . Если в обработчике установить его равным false , то ярлычок отображаться не будет. Третий параметр, передаваемый по ссылке — HintInfo . Это структура, поля которой (см. встроенную справку Delphi) содержат информацию о ярлычке: его координаты, цвет, задержки появления и т.п. В частности, имеется поле HintControl — компонент, сообщение которого должно отображаться в ярлычке, и поле HintStr — отображаемое сообщение. По умолчанию HintInfo.HintStr — первая часть свойства Hint компонента. Но в обработчике это значение можно изменить.

Приведем примеры использования событий компонента **ApplicationEvents**.

Обработчик события **OnHint**:

```
procedure TForm1.ApplicationEvents1Hint(
    Sender: TObject);
begin
    StatusBar1.SimpleText:=Application.Hint;
end;
```

отображает в полосе состояния **StatusBar1** (см. раздел 7.6) вторую часть свойства **Hint** любого компонента, в котором определено это свойство и над которым перемещается курсор мыши. Отображение происходит независимо от значения свойства **ShowHint** компонента.

Обработчик события **OnShowHint**:

```
procedure TForm1.ApplicationEvents1ShowHint(
    var HintStr: String; var CanShow: Boolean;
```

```

        var HintInfo: THintInfo);
begin
  if (HintInfo.HintControl.ClassName = 'TEdit')
  then begin
    HintStr:=(HintInfo.HintControl as TEdit).Text;
    ApplicationEvents1.CancelDispatch;
  end;

```

подменяет для окон редактирования типа **TEdit** текст ярлычков на текст, содержащийся в этих окнах редактирования. Такой прием позволяет пользователю, подведя курсор мыши к окну редактирования, увидеть во всплывающем окне полный текст, который может не быть виден обычным образом, если он длинный и не помещается целиком в окне редактирования.

Обработчик события **OnHelp**:

```

function TForm1.ApplicationEvents1Help(Command: Word;
                                         Data: Integer;
                                         var CallHelp: Boolean): Boolean;
begin
  if (Command=HELP_CONTEXT) and (Data<10)
  then begin
    Application.HelpCommand(HELP_CONTEXTPOPUP,Data);
    CallHelp:=false;
  end;
  Result:=true;
end;

```

обеспечивает отображение всех контекстных справок с номерами идентификаторов тем, меньшими 10, во всплывающем окне, не вызывая при этом WinHelp. Это дает возможность отобразить многострочные (в отличие от ярлычков) всплывающие окна, поясняющие назначение тех или иных элементов приложения.

Обработчик события **OnShortCut**:

```

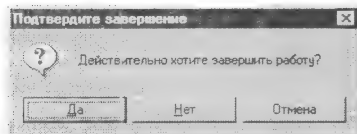
procedure TForm1.ApplicationEvents1ShortCut(
                                         var Msg: TWMKey; var Handled: Boolean);
begin
  if Msg.CharCode = Ord('Q') then
    if Application.MessageBox(
      'Действительно хотите завершить работу?',
      'Подтвердите завершение',
      MB_YESNOCANCEL+MB_ICONQUESTION)
    = IDYES
    then Application.Terminate;
end;

```

перехватывает нажатие пользователем клавиш и, если нажата клавиша с символом «Q» (в любом регистре и независимо от установки русского или английского языка), то пользователю методом **Application.MessageBox** предлагается диалоговое окно, избра-

Рис. 9.4

Диалоговое окно, отображаемое методом `Application.MessageBox`



женное на рис. 9.4. Если пользователь в нем нажмет кнопку Да, то приложение закрывается методом `Application.Terminate`.

Обработчики событий `OnActivate` и `OnDeactivate` :

```
procedure TForm1.ApplicationEvents1Activate(  
    Sender: TObject);  
begin  
    Label1.Caption:='Ура! Я работаю!';  
end;  
  
procedure TForm1.ApplicationEvents1Deactivate(  
    Sender: TObject);  
begin  
    Label1.Caption:='Увы! Меня покинули!';  
end;
```

обеспечивают появление в метке `Label1` соответствующей надписи каждый раз, когда пользователь переключается из данного приложение в другое и возвращается назад.

9.5 Объект экрана — Screen

В приложении Delphi автоматически создается глобальный объект `Screen` (экран) типа `TScreen`, свойства которого определяются из информации Windows о мониторе, на котором запускается приложение. Вы можете в любом приложении использовать, например, такие параметры объекта `Screen`, как `Height` — высота экрана и `Width` — его ширина. Так как вы используете в процессе проектирования один тип монитора, а приложение в дальнейшем может работать на мониторе другого типа, то не исключено, например, что ваша форма не поместится на экране или наоборот — будет слишком маленького размера для данного монитора. Чтобы избежать этих неприятностей, можно автоматически масштабировать свою форму, вводя, например, в обработчик ее события `OnCreate` код:

```
Form1.Width := Screen.Width div 2;  
Form1.Height:= Screen.Height div 2;
```

Этот код задает размеры формы равными половине соответствующих размеров экрана.

Разрешающую способность экрана можно определить, воспользовавшись свойством `PixelsPerInch`, указывающим количест-

во пикселей экрана на дюйм в вертикальном направлении. Это справедливо именно для вертикального направления, поскольку во многих мониторах коэффициенты масштабирования по горизонтали и вертикали различаются.

Screen имеет свойство **Forms[I]**, содержащее список форм, отображаемых в данный момент на экране, и свойство **FormCount**, отражающее количество таких форм. Вы можете использовать это свойство, например, для того, чтобы гарантировать, что на данном типе монитора размеры ни одной формы не превысят размеров экрана. Соответствующий код может выглядеть так:

```
with Screen do
  for I := 0 to FormCount-1 do
    begin
      if Forms[I].Height > Height
      then Forms[I].Height := Height;
      if Forms[I].Width > Width
      then Forms[I].Width := Width;
    end;
```

Размеры форм, превышающие размер экрана, урезаются этим кодом.

В приведенных примерах надо, конечно, предусмотреть, чтобы при изменении размеров формы адекватно изменялось и расположение компонентов на ее поверхности.

Еще одно полезное свойство объекта **Screen** — **Fonts** (шрифты). Это свойство типа **TStrings** содержит список шрифтов, доступных на данном компьютере (свойство только для чтения). Его можно использовать в приложении, чтобы проверять, имеется ли на компьютере тот или иной шрифт, используемый в приложении. Если нет — то можно или дать пользователю соответствующее предупреждение, или сменить шрифт в приложении на один из доступных, или дать пользователю возможность самому выбрать соответствующий шрифт. Например, вы можете поместить в вашем приложении компонент списка **TComboBox** и при событии формы **OnCreate** загрузить его доступными шрифтами с помощью операторов:

```
ComboBox1.Items:=Screen.Fonts;
ComboBox1.ItemIndex:=0;
```

Тогда в нужный момент пользователь может выбрать подходящий шрифт из списка. Пример использования свойства **Screen.Fonts** приведен в разделе 3.1.

В Delphi 5 в тип **TScreen** введено два новых свойства: **HintFont** и **MenuFont** типа **TFont**. Они хранят установленные в Windows шрифты отображения ярлычков и разделов меню.

Свойство **Cursor** объекта **Screen** определяет вид курсора. Если это свойство равно **crDefault**, то вид курсора при перемещении над компонентами определяется установленными в них свойствами **Cursor**. Но если свойство **Cursor** объекта **Screen** отлично от **crDefault**, то соответствующие свойства компонентов отменяются и курсор имеет глобальный вид, заданный в **Screen**. Этим можно воспользоваться для такой частой задачи, как изменение курсора на форму «песочные часы» во время выполнения каких-то длинных операций. Это можно сделать следующим образом:

```
Screen.Cursor := crHourglass;  
try  
  <выполнение требуемых длинных операций>  
finally  
  Screen.Cursor:=crDefault;  
end;
```

При успешном или аварийном окончании длинных операций курсор в любом случае возвращается в значение по умолчанию. Если в приложении в какие-то отрезки времени используется отличный от **crDefault** глобальный вид курсора, то приведенный код можно изменить, чтобы по окончании длинных операций восстановить прежнее глобальное значение:

```
var OldCursor:TCursor;  
begin  
  OldCursor := Screen.Cursor;  
try  
  <выполнение требуемых долгих операций>  
finally  
  Screen.Cursor:= OldCursor;  
end;
```

С помощью **Screen** можно получить доступ к активной в текущий момент форме вашего приложения через свойство **ActiveForm**. Если в данный момент пользователь переключился с вашего приложения на какое-то другое и, следовательно, ни одна форма вашего приложения не активна, то **ActiveForm** указывает на форму, которая станет активной, когда пользователь вернется к вашему приложению. В момент переключения фокуса с одной вашей формы на другую, генерируется событие **OnActiveFormChange**.

Аналогично с помощью свойства **ActiveControl** можно получить доступ к активному в данный момент оконному компоненту на активной форме. При смене фокуса генерируется событие **OnActiveControlChange**.

Начиная с Delphi 4 предусмотрена возможность разработки мультитрапанных приложений, работающих одновременно с мно-

жеством мониторов. При этом приложение может решать, какие формы и диалоги надо отображать на том или ином мониторе. Свойства различных мониторов, используемых в таком приложении, можно найти с помощью свойства **Screen.Monitors[I]**, где **I** — индекс монитора. Индекс 0 относится к первичному монитору. Свойство **Screen.Monitors[I]** является списком объектов типа **TMonitor**, содержащих информацию о конкретных мониторах.

Среди свойств объектов типа **TMonitor** имеются **Height** — высота и **Width** — ширина экрана монитора. Кроме того имеются свойства **Left** и **Top**. Эти свойства означают следующее. Все доступное экранное пространство можно представить себе разбитым на экраны отдельных мониторов, размещающихся слева направо и сверху вниз. Соответственно свойства **Left** и **Top** определяют координаты левого верхнего угла экрана монитора в этом логическом экранном пространстве. Объекты типа **TMonitor** имеют также свойство **MonitorNum** — номер монитора, представляющий собой его индекс в свойстве **Screen.Monitors[I]**.

Для управления тем, на каком мониторе должна появляться та или иная форма, служит свойство формы **DefaultMonitor**. Это свойство может принимать значения:

dmDesktop	не предпринимается попыток разместить форму на конкретном мониторе
dmPrimary	форма размещается на первом мониторе в списке Screen.Monitors
dmMainForm	форма появляется на том мониторе, на котором размещена главная форма
dmActiveForm	форма появляется на том мониторе, на котором размещена текущая активная форма

Научно-популярное издание

Архангельский Алексей Яковлевич

100 компонентов общего назначения библиотеки Delphi 5
(Все о Delphi)

Оформление *В.И. Череховского*
Компьютерная верстка *К.А. Свиридова*

Подписано в печать 28.09.99. Формат 60х84 $\frac{1}{16}$. Усл. печ. л. 15,81.

Бумага газетная. Печать офсетная.

Тираж 5000 экз. Заказ 4827

ЗАО «Издательство БИНОМ», 1999 г.
103473, Москва, Краснопролетарская, 16

Лицензия на издательскую деятельность № 065249 от 26 июня 1997 г.

Отпечатано с готовых диапозитивов в полиграфической фирме
«Красный пролетарий»
103473, Москва, Краснопролетарская, 16.

100 компонентов общего назначения библиотеки Delphi 5

В книге рассмотрено 100 компонентов и классов Delphi 5 - практически все компоненты общего назначения: ввод и вывод текстовой, цифровой, графической и мультимедиа информации, управляющие элементы, дизайн и организация прикладных программ. Проведен сравнительный анализ функций и возможностей компонентов, даны рекомендации по их применению. На многочисленных примерах показано, как удачное использование различных компонентов позволяет сделать программу эстетически привлекательной и удобной для пользователя.

ВСЕ О DELPHI это:

**все аспекты применения
Delphi**

**сведения по новым
версиям Delphi
по мере их выпуска**

**методика, обзоры и
справочные данные**

**Для всех пользователей:
от начинающих до
профессионалов**

ISBN 5-7989-0154-8



9 785798 901548

ВСЕ О DELPHI